ISSN 0265-2919

90p 62

THE HOME COMPUTER ADVANCED COURSE

MAKING THE MOST OF YOUR MICRO

An ORBIS Publication

IR£1.15 Aus $2.15 NZ $2.65 SA R2.45 Sing $4.50

CONTENTS

APPLICATION

ON THE BRAIN We introduce a series of articles on the major aspects of contemporary research into artificial intelligence. First, though, we look back at the earliest work of AI researchers — 1221

HARDWARE

FINDING ITS NICHE The Wren Executive is a sturdy portable computer with both built-in modem and monitor. Supplied with CP/M and BBC BASIC, the machine may prove more attractive to educational users — 1229

SOFTWARE

THE RIGHT SORT To conclude our series on databases, we provide answers to some of the questions a potential user may ask — 1224

COMPUTER SCIENCE

DYNAMIC DUO As our PASCAL tutorial course approaches its conclusion, we show how the dynamic allocation of memory provides the language with greater flexibility — 1234

JARGON

FROM PAPER TAPE TO PARALLEL PROCESSING A weekly glossary of computing terms — 1228

PROGRAMMING PROJECTS

MUTINY ON THE BOUNTY There is always the possibility that the crew will become discontented with the progress of the voyage if things start going badly. The module we develop for our simulation game this week concerns the eventuality of a mutiny — 1232

MACHINE CODE

BEAT GENERATION A program to turn the Commodore 64 into a drum machine — 1237

WORKSHOP

THE CURVATURE OF THE SPLINE We discuss a software technique to enable our robot arm to smoothly wave goodbye — 1226

REFERENCE CARD Another part of the Commodore 64's memory map — INSIDE BACK COVER

Next Week

- **We begin a new Workshop project to build a digital tracer for the BBC Micro.**
- **The Tandy Corporation has recently introduced an IBM-compatible machine priced at around £1,100. We examine this new computer and test the manufacturer's claim of 'complete compatibility'.**

QUIZ

1) What was Rosenblatt's Perceptron?
2) What do you understand to be a 'query language'?
3) How many Kbytes of memory can be addressed by the Wren Executive's Z80B chip?
4) What purpose does the eighth hole perform on a paper tape system?

Answer's To Last Week's Quiz

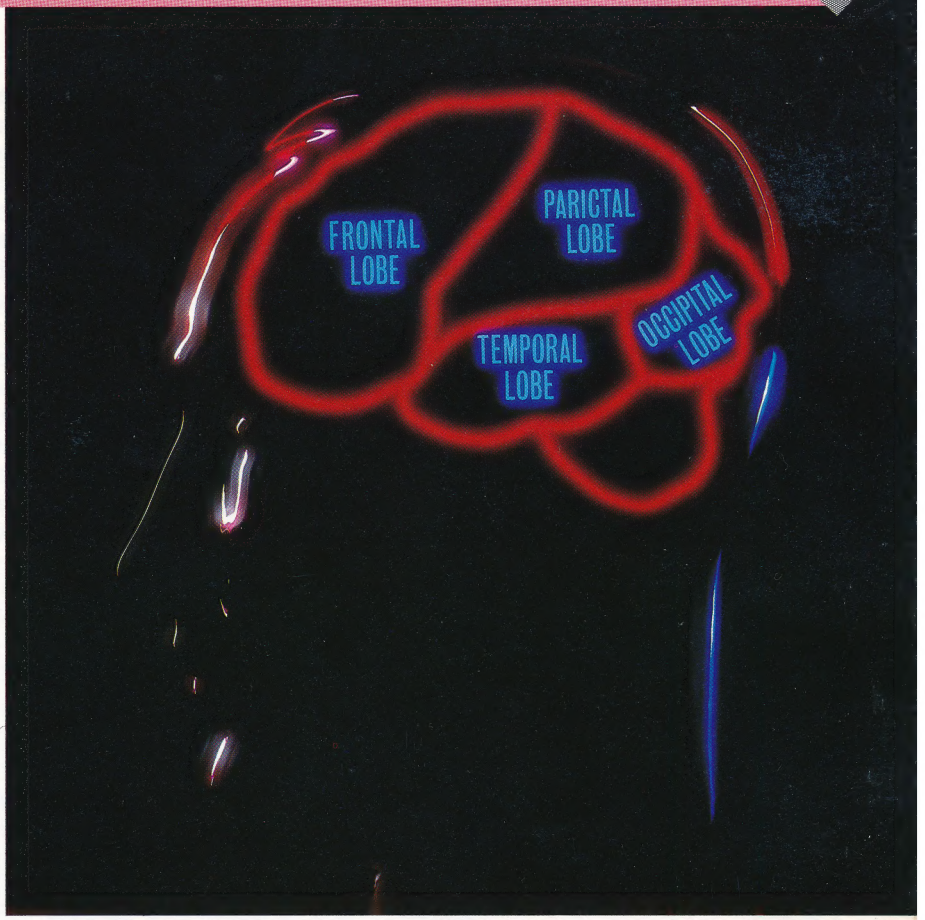
1) Galvanic Skin Resistance (GSR) is based on the principle that electrical conductivity of the skin varies with emotional stress.
2) Both CP/M and Dr LOGO are products of Digital Research.
3) The VIC-II chip sets the bus available line to low, which tells the processor to finish what it's doing and concentrate on feeding data to the video chip.
4) Although we can specify the hi-byte of a program's start address, we cannot state what the lo-byte will be (because we do not know the current state of the data in memory). Therefore, we create a 256-byte table to specify all the possibilities for the lo-byte.

Editor Stephen Cooke; **Deputy Editor** Steve Colwill; **Art Editor** Claudia Zeff; **Production Editor** Bobby Pickering; **Designer** Julian Dorr; **Art Assistant** Liz Dixon; **Staff Writer** Steve Malone; **Sub Editor** Jon Kaye; **Contributors** Dr Andrew Bangham, Nick Walsh, Dr Antonia Jones, Terry Barrett, Steve Malone, David Mudd, Anthony Ginn, Steve Colwill; **Software Consultants** Pilot Software City; **Group Art Director** Perry Neville; **Managing Director** Stephen England; **Published by Orbis Publishing Ltd; Editorial Director** Brian Innes; **Project Development** Peter Brookesmith; **Executive Editor** Maurice Geller; **Production Assistant** Alastair Gourlay; **Subscription Manager** Christine Allen; **Designed and produced by Bunch Partworks Ltd; Editorial Office** 14 Rathbone Place, London W1P 1DE; **© APSIF Copenhagen 1985; © Orbis Publishing Ltd 1985; Typeset by Universe; Reproduction by Mullis Morgan Ltd; Printed in Great Britain by Heanor Gate Printing Ltd, Derby**

HOW TO OBTAIN ISSUES AND BINDERS FOR THE HOME COMPUTER ADVANCED COURSE – Issues can be obtained by placing an order with your newsagent or direct from our subscription department. If you have any difficulty obtaining any back issues from your newsagent, please write to us stating the issue(s) required and enclosing a cheque for the cover price of the issue(s). **AUSTRALIA** – please write to: Gordon & Gotch (Aus) Ltd, 114 William Street, PO Box 767G, Melbourne, Victoria 3001. **MALTA, NEW ZEALAND & SOUTH AFRICA** – Back numbers are available at cover price from your newsagent. In case of difficulty, write to the address given for binders.
UK/EIRE – Price: 90p/IR£1.15. Subscription: 6 months: £26.00. 1 Year: £52.00. Binder: please send £3.95 per binder, or take advantage of our special offer in early issues. **EUROPE** – Price: 90p. Subscription: 6 months air: £44.72. Surface: £36.14. 1 year air: £89.44. Surface: £72.28. Binder: £5.00. Airmail: £8.25. **MALTA** – Obtain binders from your newsagent or Miller (Malta) Ltd, MA Vassalli Street, Valetta, Malta. Price: £3.95. **MIDDLE EAST** – Price: 90p. Subscription: 6 months air: £50.18. Surface: £36.14. 1 year air: £100.36. Surface: £72.28. Binder: £5.00. Airmail: £8.25. **AMERICAS/ASIA/AFRICA** – Price: US/CAN$1.95/90p. Subscription: 6 months air: £59.54. Surface: £36.14. 1 year air: £119.08. Surface: £72.28. Binder: £5.00. Airmail: £9.50. **SOUTH AFRICA** – Price: SA R2.45. Obtain binders from any branch of Central News Agency or Intermag, PO Box 57394, Springfield 2137. **SINGAPORE** – Price: Sing $4.50. Obtain binders from MPH Distributors, 601 Sims Drive, 03-07-21, Singapore 1438. **AUSTRALASIA/FAR EAST** – Price: 90p. Subscription: 6 months air: £64.22. Surface: £36.14. 1 year air £128.44. Surface: £72.28. Binder: £5.00. Airmail: £9.75. **AUSTRALIA** – Price: AusS2.15. Obtain binders from First Post Pty Ltd, 23 Chandos Street, St Leonards, NSW 2065. **NEW ZEALAND** – Price: NZS2.65. Obtain binders from your newsagent or Gordon & Gotch (NZ) Ltd, PO Box 1595, Wellington.
ADDRESS FOR BINDERS AND BACK ISSUES – Orbis Publishing Limited, Orbis House, Bedfordbury, London WC2 4BT. Telephone 01-379 5211. Cheques/postal orders should be made payable to Orbis Publishing Limited. Binder prices include postage and packing and prices are in stirling. Back issues are sold at the cover price, and we do not charge carriage in the UK.
NOTE – Binders and back issues are obtainable subject to availability of stocks. Whilst every attempt is made to keep the price of the issues and binders constant, the publishers reserve the right to increase the stated prices at any time when circumstances dictate. Binders depicted in this publication are those produced for the UK and Australian markets only. Binders and Issues may be subject to import duty and/or local taxes, which are not included in the above prices unless stated.
ADDRESS FOR SUBSCRIPTIONS – Orbis Publishing Limited, Hurst Farm, Baydon Road, Lambourn Woodlands, Newbury Berks, RG16 7TW. Telephone: 0488-72666. All cheques/postal orders should be made payable to Orbis Publishing Limited. Postage and packaging is included in subscription rates, and prices are given in sterling.

COVER PHOTOGRAPHY BY PAUL CHAVE

ON THE BRAIN



PAUL CHAVE

The field of artificial intelligence unquestionably represents one of the most exciting aspects of computing today. We begin a series of articles exploring this fascinating subject in depth, and we will be including example programs in BASIC to show how key AI techniques can be applied to your home micro.

In the early 1970s, Artificial Intelligence (AI) was in the doldrums, especially in the UK. It was widely regarded as being the lunatic fringe of computer science, and an influential report for the Science Research Council by Sir James Lighthill had just recommended a drastic cut-back in funding. Today, AI is riding high. Its practitioners are suddenly so respectable that venture capitalists are seeking them out with tempting offers. Government agencies are backing costly R & D programmes for fear of falling behind in the AI race. Meanwhile, software vendors are distributing glossy press releases re-defining their products as AI systems.

To understand where AI is today and where it is likely to go in the future, it is useful, as with so many technologies, to look at its past. We can divide our condensed history of artificial intelligence into four decade-sized chunks, each characterised by a dominant theme. This necessarily oversimplifies things but, in doing so, brings out the major points. Each of the themes can be thought of as the answer you would have likely received had you asked an AI researcher of the time: 'What is artificial intelligence all about?'

1950s Neural Nets
1960s Heuristic Search
1970s Expert Systems
1980s Machine Learning

In 1943, Warren McCulloch and Walter Pitts proposed a model of the neuron in the human and animal brain. These abstract nerve cells provided the basis for a mathematically symbolic representation of brain activity. Other workers, notably Norbert Wiener, elaborated these and similar ideas into the field that became known as 'cybernetics' (a system based on the premise that a sentient machine can be constructed by using biological methods of feedback and analysis as a model). It was from cybernetics that AI emerged in the 1950s.

Early AI researchers took McCulloch's formalised neuron as their building block. Considering the immense complexity of the brain, it was not altogether surprising that they failed to generate intelligent systems based on this model. In effect, they were saying: 'The brain is an intelligent problem solver, so let's simulate the brain'. But the early hardware, to say nothing of the software, was not up to the job.

One of the few successful systems of the time was Rosenblatt's Perceptron. It was an elementary visual system that could be taught to recognise patterns. Shown in the diagram, the Perceptron consists of a finite grid of light-sensitive cells, modelling a miniature retina. In addition, there are a number of feature-detecting elements — graphically called 'demons' — which monitor the state of groups of cells in the grid. They respond, when characteristic subpatterns are present, by sending a signal to a decision-maker. The decision-maker multiplies each signal from a local demon by a positive or negative weighting factor and the resulting numbers are added. If the total exceeds a set threshold, the Perceptron indicates: yes; otherwise it will be no. Thus one Perceptron can distinguish between two classes of images,

Mechanical Minds
Artificial Intelligence is concerned with the design of computer systems that perform tasks which would require intelligence if done by a human. However, its scope has been broadened to include perceptual tasks, such as sight and hearing. The main concern of AI research is to program machines so that they imitate aspects of human behaviour and understanding

though the same principles can be extended to deal with more than two.

There were at one time great hopes that Perceptrons could tackle a wide range of problem-solving tasks, but these soon evaporated. AI workers then began to regard human thinking as a co-ordination of essentially simple symbol-manipulating tasks. Although this was ostensibly a great change in the direction of research, the designers were at least on firmer ground, since computers can do things such as searching, comparing symbols and so on, which they identified as being the foundations of intelligent problem solving. The difficult part was putting these simple activities together.

The most influential workers in the 1960s were Alan Newell and Herbert Simon of Carnegie-Mellon University, who worked on theorem-proving and computer chess, among other investigations. Their most impressive achievement was a program called GPS, the General Problem Solver. It was general insofar as the user defined a 'task environment', in terms of the objects of a particular domain and the operators that could be applied to those objects. However, this generality was restricted to puzzles with a relatively small set of states and well-defined rules. It could work on the Towers of Hanoi (see page 474), Cryptarithmetic and other similar types of problems in which formalised micro-worlds represented the parameters within which problems could in fact be solved. What the GPS could not do was solve what people would regard as real-world problems, such as making medical diagnoses, or business decisions based on fluctuating currencies.

The central idea behind GPS was that problem solving was a search through a space of potential solutions. To make the search efficient, it had to be guided by heuristic (learning from its own discoveries) rules that directed it towards the desired destination, finding a path through, essentially, trial and error. Thus an automaton wandering around a maze would have to use an exhaustive search technique if it knew nothing about the structure of that maze; but if it had some way of telling when it was getting 'warm', it could be expected to reach its goal state sooner (but not always, since heuristics is not guaranteed to work, and occasionally may lead to sojourns down blind alleys). During this period, AI workers devised several heuristically guided search strategies, which we will consider in subsequent instalments.

The General Problem Solver, as we have said, was not much good at solving real-life problems. In the 1970s a team led by Edward Feigenbaum at Stanford University began to remedy that defect. Rather than trying to computerise general intelligence, they focused on very narrow areas of expertise. Thus the expert system was born.

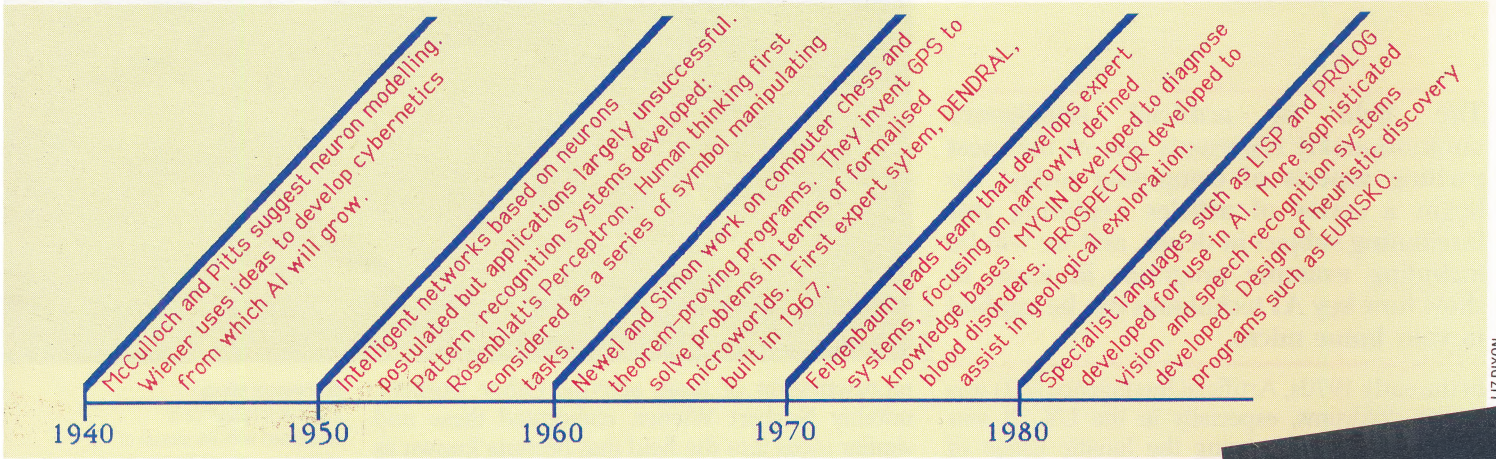
The first expert system was Dendral, a mass-spectrogram interpreter built as early as 1967; though the most influential has proved to be Mycin, which dates from 1974. Mycin diagnoses bacterial infections of the blood, and prescribes drug therapy. It has spawned a whole family of medical diagnostic 'clones', some of which are in routine clinical use. For instance, Puff, a lung function diagnostic tool based on the Mycin plan, is routinely employed at the Pacific Medical Center near San Francisco.

Mycin introduced a number of new features that have become the hallmarks of the expert system. First, its 'knowledge' consists of hundreds of rules such as the following:

RULE No. 47:
IF 1) the site of the culture is blood, and
2) the identity of the organism is not known with certainty, and
3) the stain of the organism is gramneg, and
4) the morphology of the organism is rod, and
5) the patient has been seriously burned
THEN there is weakly suggestive evidence (0.4) that the identity of the organism is pseudomomas.

Secondly, these rules are probabilisitic. Shortliffe, the inventor of Mycin who was a medical doctor, devised a scheme based on certainty factors to allow the system to reach plausible conclusions from uncertain evidence. Thus the number 0.4 is not strictly speaking a probability: essentially it is a 'fudge factor'. The significant point, however, is

Historical Perspective
AI is becoming an increasingly important area of practical research, with applications in many fields. The history of AI, however, covers a short period of time compared with many other scientific fields, as our diagram shows . . .

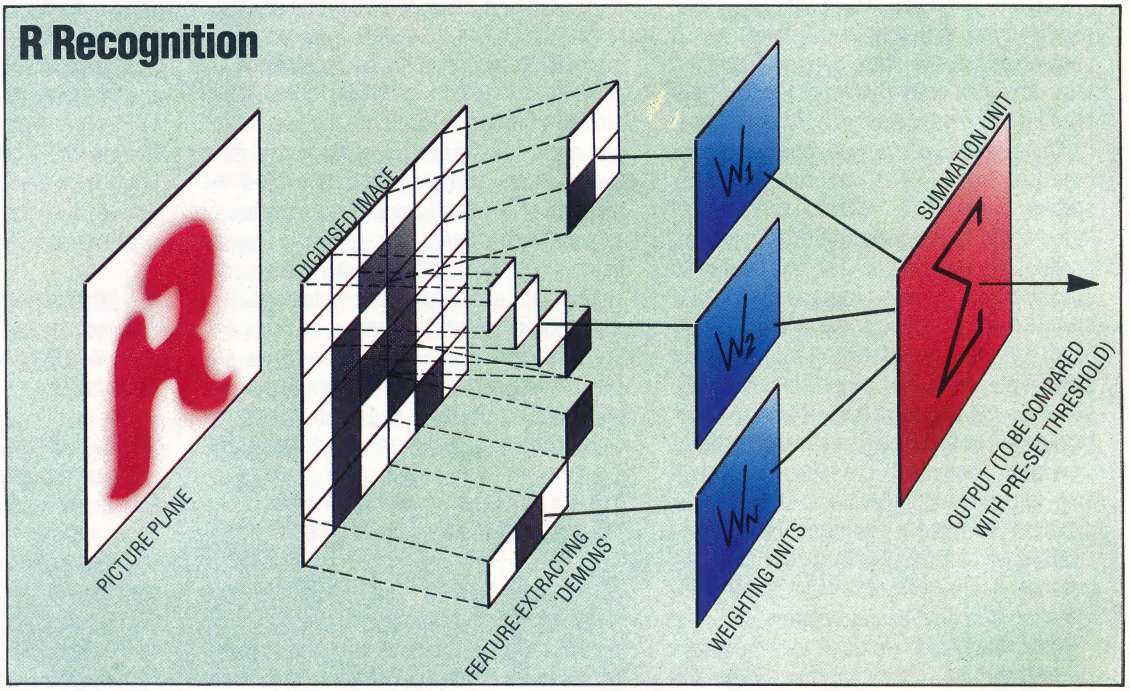

McCulloch and Pitts suggest neuron modelling.
Wiener uses ideas to develop cybernetics from which AI will grow.
Intelligent networks based on neurons postulated but applications largely unsuccessful.
Pattern recognition systems developed.
Rosenblatt's Perceptron. Human thinking first considered as a series of symbol manipulating tasks.
Newel and Simon work on computer chess and theorem-proving programs. They invent GPS to solve problems in terms of formalised microworlds. First expert sytem, DENDRAL, built in 1967.
Feigenbaum leads team that develops expert systems, focusing on narrowly defined knowledge bases. MYCIN developed to diagnose blood disorders. PROSPECTOR developed to assist in geological exploration.
Specialist languages such as LISP and PROLOG developed for use in AI. More sophisticated vision and speech recognition systems developed. Design of heuristic discovery programs such as EURISKO.
1940
1950
1960
1970
1980
LIZ DIXON

COURTESY OF COMPUTER WEEKLY

Edward Feigenbaum
As leader of an AI team at Stanford University, California, Edward Feigenbaum developed the first expert systems. These were notable because they moved away from the idea of general intelligence programs to systems operating within closely defined limits of knowledge and objectives

that Mycin and systems like it can arrive at correct conclusions even with incomplete and partly erroneous information. They employ some method of approximate reasoning — whether based on probabilities, Fuzzy Logic (see page 676), certainty factors or some other likelihood calculus—for deriving a good estimate of the truth from an imperfect collection of data.

Thirdly, Mycin can explain its own reasoning process. The physician using it can interrogate it in various ways, either to ask how it reached a particular conclusion or why it is requesting a certain item of information. The system answers by retracing and describing the deductive process that led to the current state. This degree of user-friendliness was essentially a by-product of the rule-based style of programming.



Perceptron Perceptions
The image to be sampled (in this case, the letter R) is projected onto a plane and digitised. 'Demons' sample a small group of pixels (e.g. 4 at a time) and respond if the pattern they are programmed to recognise is present. Each demon's response (either 0 or 1) is multiplied by a weighting factor, depending on its importance within the pattern as a whole, and the weighted responses are added together. This result is then compared to a threshold value. If it is greater than the threshold, then the system recognises the original shape; otherwise it does not. The threshold and demon weightings can be adjusted when training the Perceptron to recognise a particular shape

Finally, and crucially, Mycin *works*. It does what a human requires years of training for. In fact, Mycin is used more in teaching than in diagnosing, but the point is that large corporations, governments and the media are all growing interested. It is early yet but corporate America has scented the sweet smell of profits. AI is losing its innocence.

THE STATE OF THE ART

This brings us up to the 1980s. Expert systems are in fashion and their 'active ingredient' is knowledge — for the scope and quality of its knowledge base determines the success of an expert system. But knowledge is not something you can squeeze into a computer program like toothpaste from a tube. Codifying a human expert's skill can be a long and labour-intensive process. So while the world is marvelling over expert systems, artificial intelligence has moved on to concentrate on the problem of machine learning — which is one way of acquiring knowledge automatically. AI has always been a moving target, and at the centre of that target right now is a program called Eurisko.

Eurisko is a discovery program that extends and improves its own body of heuristic rules automatically, by induction. Apart from winning the Trillion Credit Squadron naval wargame three years in succession (despite rule changes intended to prevent it) Eurisko has also been applied to practical problems. One result was the invention of a novel 3-dimensional logic gate in the field of integrated-circuit design. There can be little doubt that systems like Eurisko represent the leading edge of AI research. And since AI is itself the leading edge of computing science, this is the place to look for a peek into the future of computing.

Ironically enough, by concentrating once again on learning, AI has returned to its roots, because learning was seen as the key problem in the early cybernetic days.

This series of articles represents a practical DIY guide to artificial intelligence. Over this time we will be covering the major problem areas of AI (such as vision and natural language processing) as well as the techniques that have been developed to deal with them. As usual, all of the example programs will be presented in BASIC, and the BBC Micro, Commodore 64 and the Sinclair Spectrum will be represented in our Basic Flavours boxes.

Although the well-equipped AI investigator may have a room full of machines running from LISP, we can only expect to be using a home micro with a BASIC interpreter. Nevertheless, we hope to illustrate some of the key AI techniques that can be applied to your machine — it is, after all, as powerful as a mainframe of twenty years ago. You needn't worry about not having access to LISP or PROLOG, which are the major AI programming languages, because once you understand the methods involved, you will be able to implement them in any language.

THE RIGHT SORT

Q When shopping around for a database manager, what are the major points I should keep in mind?

A When using a word processor, the time taken to search files or to move blocks of characters around (equivalent to sorting) is an irritating overhead. However, once performed, these activities are not usually needed again. The 'search and sort' overhead is therefore a relatively small percentage of the total program running time. It is data entry (typing in the text) and printing that tend to take up most of the time.

This situation is reversed, however, in the case of DBMs. The records, generally speaking, are in effect very small 'documents' that require little time to type in. Sorting the records and searching them to extract those required are what a DBM spends most of its time doing when it is being used. If the data file is larger than what can fit completely in RAM, as is highly likely, extensive reading from and writing to mass storage will be required. The shortcomings of slow and 'sequential' cassette storage will then become highly evident. Unless your database is limited in size and importance — a catalogue of a small record collection, for instance — you will really have to consider investing in a disk-based system.

Most databases are used in a business context of one sort or another, where the 'time is money' philosophy applies. In other words, if you really need a DBM, you will require a disk-based system, although Archive on the QL's microdrives is an adequate alternative in the interim.

Q How many records should my DBM be capable of handling, and is it worthwhile having one with more records than I really need?

A Your database application may or may not require the ability to handle a large number of records. But try to imagine the largest number of records you would ever need, double it, and then look for a DBM that can handle that many. If your database is for stock control and inventory, and you do not carry many parts, you may not need a DBM that can handle 32,000 records. But if you are using the DBM to catalogue a library, 32,000 records might not be enough. DBMs able to handle more than 64,000 records are fairly rare and, anyway, would normally need powerful computers with very large hard disk memories.

Q If I'm not certain about the number of fields I will require, how can I judge what DBM to buy?

A Most DBMs allow only a certain number of fields to be contained in any record. It is often difficult to anticipate how many fields will actually be required. Most allow more than enough fields, though fields are often limited in length to one line, which can make multi-line text fields awkward to use. A more common limitation than the number of fields allowed is the number of allowable characters per record. A typical figure might be 1,020 characters per record enabling the user to choose how to divide this allocation between the number of fields and the character length of each field. In any event, make sure that the maximum number of characters per field and fields per record will allow you to build the records you want.

Q Are key fields an important aspect and, if so, how many will I need?

A A key field is one that can be searched or otherwise manipulated by the DBM. If a field is not designated as being key, it cannot be used to extract records. Here are a couple of examples of how typical key fields might be used:

```
SEARCH SUPPLIER FOR 'Timson
Engineering Ltd'
```

The key field SUPPLIER in every record is being searched for a match with the specified character string, or:

```
SEARCH PRICE >= 45.50
```

The key field PRICE is being searched for figures greater than or equal to 45.50. Some DBMs allow all, or any number of fields to be specified as key fields when the record 'skeleton' is created; others allow only a certain maximum number of fields to be so designated. As a rule of thumb, the more fields that can be designated as key fields the better.

Q Will I be restricted to the language supplied with my computer or will I be able to program the DBM myself, without it?

A Some DBMs include a built-in programming language allowing sophisticated sequences of activities to be performed automatically. In a business environment, these languages enable programs to be written that can save a great deal of work on the operator's part.

Two outstanding examples are dBase II and Archive. If your database requirements will have set ways of dealing with data, such as 'print out all today's sales followed by a list of all parts in stock and a list of all parts below the "re-order" threshold', then a DBM with a built-in programming language is something to look for. If, on the other hand, you do not know from day to day just how the database will be used, then the usual 'query language' commands will suffice.

Q What should I be looking for if the records I will be creating need to be related to data other than what's in those records?

A The simpler database managers can work on only one file of records at a time. For many applications, this is perfectly adequate. Far more useful, however, is the ability of a DBM to work on two or more files simultaneously. The classic situation where this is helpful is a stock control database in which each part can have more than one supplier. We would have one database file for 'parts' and another separate file for 'suppliers'. It is extremely useful, if you make a query on one file, to be able to make queries on other, related files.

To illustrate the point, let's suppose we ran an antique shop and kept a 'stock' database file on what we had in stock. We might also keep a record of our suppliers in which we noted what they have available but have not yet bought. Our records for DESCRIPTION 'FURNITURE, SWEDISH' might have no entries, but a cross-reference to your SUPPLIERS file might reveal KURT JAKOBSEN ANTIK, GAMLA STAN 56, STOCKHOLM. His record might reveal:

```
DESCRIPTION:    SOFA
MAKER:          ANDERSEN
DATE:           1824
PRICE SKR:      12000

DESCRIPTION:    SOFA
MAKER:          GRIMM
DATE:           1874
```

PRICE SKR: 9800

and countless other entries of items available. In other words, when a single file will not suffice, referencing to one or more other files may be what is needed.

Theoretically, it is always possible to have enough spare fields within any particular file to allow for any additional information that may be required. In practice, this may not be possible. Suppose Mr Jakobsen telephones us to say that he has just received two lovely 19th century sofas today that we might be interested in. It is obviously very much easier to enter the details under JAKOBSEN's record than to go through all the DESCRIPTION fields in the STOCK file and enter Mr. Jakobsen's new acuisitions.

To simplify, whenever you have a situation where there's more than a one-to-one relationship between parts of a record in a database to other data, a multi-file DBM is the one primarily to consider.

Q Dependent and calculated fields seem rather useful. How can they help me with the work I'll be doing on the DBM?

A A refinement offered by some DBMs is known as dependent fields. These are fields that appear (during data entry and subsequently) only if they are needed. For example, a database that has a field NO. OF CHILDREN might display no extra fields if the entry is 0, but NAME OF FIRST CHILD, AGE OF FIRST CHILD and so on if the entry was 1, and so on.

Dependent fields are, however, just a refinement and seldom, if ever, essential. Some DBMs also allow the data in certain fields to be used as arguments in arithmetical operations — almost like a mini-spreadsheet. Such systems would allow, for example, the number of items in a NO. IN STOCK field to be multiplied by the figure in the PRICE field in each record, as well as summing up all the results for the records in a file to give a total value in stock result. This kind of facility almost always goes with a built-in programming language and can be extremely useful.

Q What are the differences between menu-driven and command-driven DBMs, and is one more suitable than the other?

A A menu-driven database manager is one that presents options on the screen at each stage of activity. A command-driven program expects you to learn its vocabulary of commands (often involving a Control key-plus-letter combination of key presses) to make it work. Although menu-driven programs tend to be easier for the beginner, command-driven ones are usually quicker to use once you have learnt the commands. Provided that the commands are chosen logically (S for SEARCH, P for PRINT and so on) command-driven software is likely to be easier to use in the long run.

Q What sorts of facilities are built into DBMs that will check for errors and prevent erroneous entries?

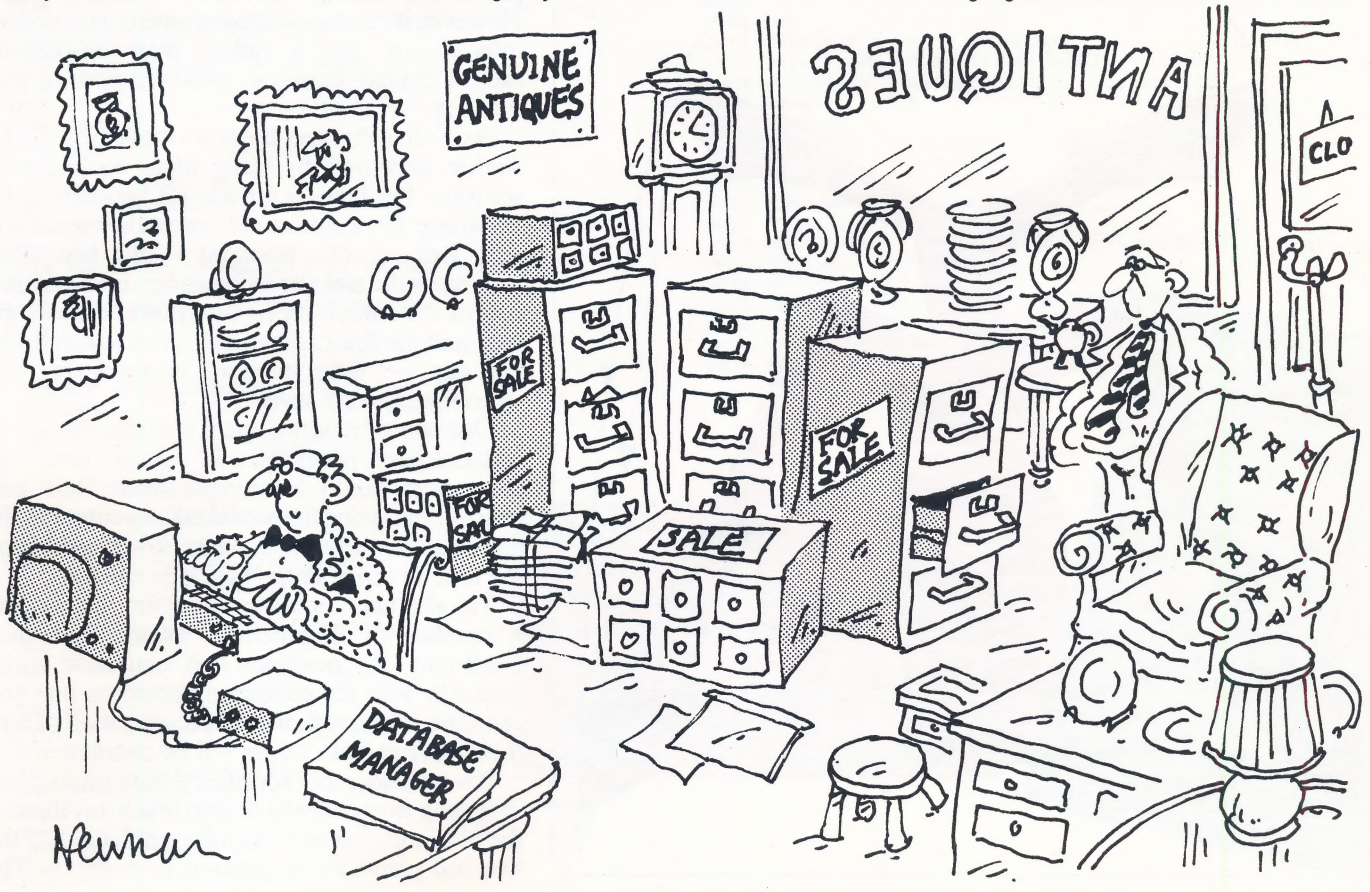
A A well designed DBM should allow you to specify what type of data may be entered into any field, thereby helping to reduce erroneous entries. For example, 31/02/85 should be automatically rejected, as should an entry such as:

PRICE: Oil flow meter mounting flange

the best software always includes the facility for placing limits on the type of data allowable; on the other hand, over-rigid checking can sometimes be a nuisance. In any event, check what data validation facilities are offered.

Q All that being said, what DBM should I buy?

A Recommending a best buy for a DBM is all but impossible; you have to know what you will use it for before you can buy one. However, we would recommend for all but the least sophisticated applications, a DBM that is disk–based, has multi-file access and has a built-in programming language in addition to the normal query language.



THE CURVATURE OF THE SPLINE

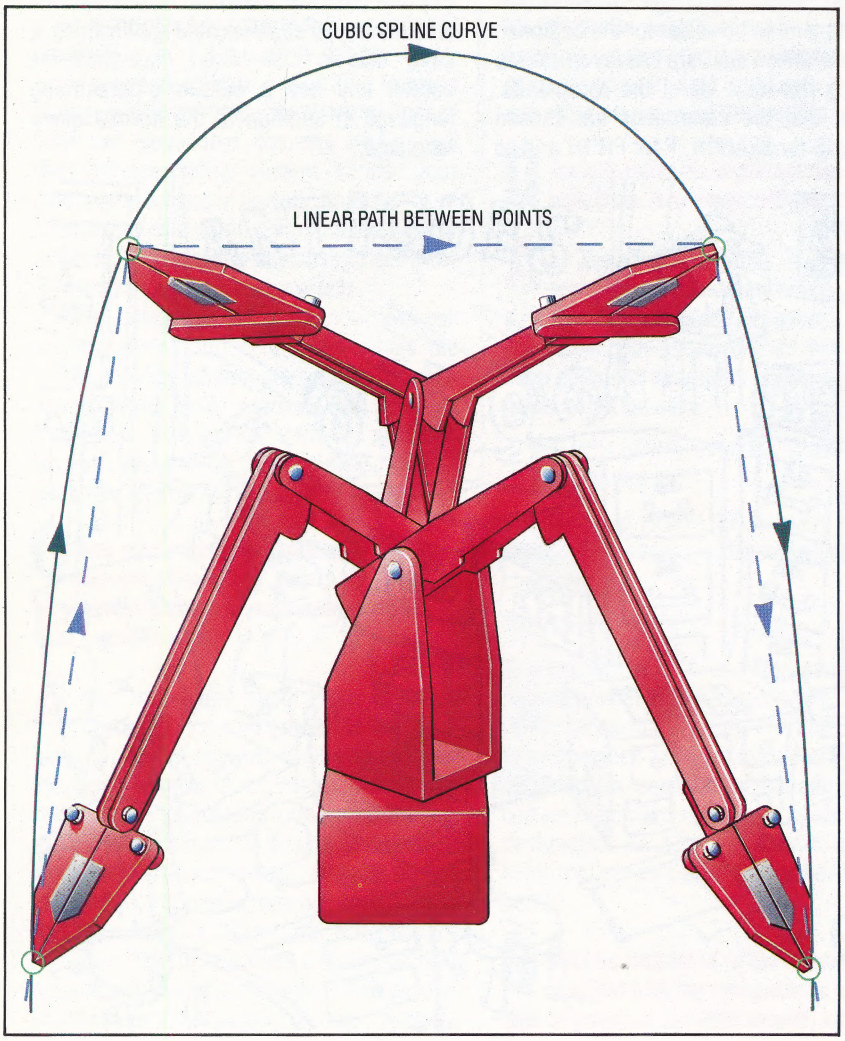
In this final instalment of the Workshop robot arm series, we review the software already developed for the arm. We also consider further developments that might be made to improve the smoothness of the arm's movements by using a mathematical process to produce 'cubic splines'.

The moveservo routine developed as part of the arm controlling software (see pages 1175 and 1195) reduces the jerkiness inherent in the arm as it moves from one position to another. The motors receive their angular information from a series of locations in memory, beginning at ANGLE.

To improve the smoothness of the arm's movement, as the motors take up new positions in response to changes in the ANGLE locations, the values are not POKEd directly into these locations, but into a group of corresponding locations labelled NEWPOS. The moveservo routine, when called, compares the values in the corresponding locations in ANGLE and NEWPOS and lowers or increases ANGLE as appropriate, one unit at a time. It repeats the process until the values in NEWPOS and ANGLE are the same.

By POKEing angle values from BASIC into NEWPOS rather than ANGLE, we effectively buffer the motors against sudden, large changes in the angle. The task of moving a cotton reel from a known position into a cup is now relatively easy. The arm can be taken through the moves slowly using the appropriate keys on the keyboard, and key positions on the curve can be saved in an array. The information in the array can then be replayed at any speed by inserting a delay factor in moveservo and stepping through the array.

Smoothing Out The Rough Edges
If four crucial positions for the arm are saved when using the arm programmer program then, when replaying the sequence, the arm will move linearly between the points (shown by the dotted line). However, it is possible to calculate a number of intermediate points that generate a smooth curve (known as a 'cubic spline') passing through the four saved positions. The co-ordinates of points on this curve can be used to make the arm sweep more smoothly through its programmed path



KEVIN JONES

AVOIDING DISCONTINUITY

Further improvements are possible. To a first approximation, all the angular movements are proportional to the changes in NEWPOS, depending on how far from the pivot each linkage is fastened compared with the radius of the motor drive lever. The moveservo routine therefore moves each part of the arm smoothly and with constant angular velocity (the degrees of rotation per second) to its new position. Whenever it reaches a new position, however, the arm may have to suddenly change direction as it starts to make its way towards the next saved position in the sequence; in other words, it hits a discontinuity.

This problem is not so bad as the original problem of making sudden large changes of angle. However, it can be avoided by saving a number of intermediate points rather than introducing radical changes between one saved position and the next.

As is often the case, the arm's movement can be further improved by using more sophisticated software, but there is a trade-off between speed and space. If the cotton reel is moved by specifying only four separate positions — the first when picking up the reel, the second when lifting the reel up, the third when taking up a position above the cup, and the fourth when the reel is lowered into the cup — little memory is used, but the discontinuities are severe.

One way of reducing them is to specify, say, 60 intermediate positions, in which case the movement would be comparatively fluid and elegant. But doing this would take up substantially more memory to store the data, to say nothing of the time to input all the positions manually.

The alternative requires taking time to calculate a number of intermediate values, and then combining the positions that you have saved manually with the connecting positions that you calculate. The question now presents itself of how these intermediate values will be calculated.

If we consider four specified points, then a piece of sprung steel could be bent to touch any three of the four points at any one time, particularly if the two end points are in convenient positions. The

mathematical equivalent of bending a piece of sprung steel is to interpolate intermediate points using a 'cubic spline' (so-called because it includes values raised to the power three).

The program given here demonstrates how intermediate points that fit a cubic curve can be interpolated from the four points specified. The resulting movement between the points will be a smooth sweeping curve; a considerable improvement over linear interpolation. To use this method for a four motor system, such as our Workshop robot arm, a separate cubic spline would need to be interpolated mathematically for each motor.

Cubic Splines

BBC Micro:

```
1000 REM ************************
1010 REM ** BBC CUBIC SPLINES **
1020 REM ************************
1030:
1040 MODE 0
1050 VDU 23,240,24,126,66,219,219,66,126,24
1060 V=7
1070 DIM X(V+3),Y(V+3),M(V+3),Z(V+3)
1080 X(2)=0 :X(3)=10 :X(4)=40 :X(5)=50
1090 Y(2)=2 :Y(3)=12 :Y(4)=12 :Y(5)=2
1100 REM linearly extend the ends
1110 X(1)=X(2)-(X(3)-X(2))
1120 Y(1)=Y(2)-(Y(3)-Y(2))
1130 X(6)=X(5)+(X(5)-X(4))
1140 X(7)=X(6)+(X(6)-X(5))
1150 Y(6)=Y(5)+(Y(5)-Y(4))
1160 Y(7)=Y(6)+(Y(6)-Y(5))
1170 CLG:VDU 5:MOVE 100,700
1180 PRINT"Demonstration of a routine"
1190 PRINT"which incorporates a cubic spline"
1200 PRINT"joining knots"
1210 FOR I=2 TO 5
1220 MOVE X(I)*20-4,Y(I)*20+15:PRINT CHR$(240);
1230 NEXT
1240 GOSUB 1500:REM SET AKIMA
1250 FOR X=X(2) TO X(5)
1260 GOSUB 1330:REM AKIMA FUNCTION
1270 IF X=X(2) THEN MOVE X*20,Y*20
1280 IF X<>X(2) THEN DRAW X*20,Y*20
1290 NEXT
1300 END
1310 :
1320 :
1330 REM **** AKIMA FUNCTION ****
1340 REM AKIMA RUCKDESCHEL BOOK 2  (BYTE/ MCGRAW-HILL)
1350 N=1
1360 REM CHECK TO SEE IF X IS IN THE TABLE RANGE
1370 IF X<X(1) OR X>X(V-2) THEN RETURN
1380 REM FIND RELEVANT TABLE INTERVAL
1390 I=0
1400 I=I+1:IF X>=X(I) THEN 1400
1410 I=I-1
1420 REM BEGIN INTERPOLATION
1430 B=X(I+1)-X(I)
1440 A=X-X(I)
1450 Y=Y(I)+Z(I)*A+(3*M(I+2)-2*Z(I)-Z(I+1))*A*A/B
1460 Y=Y+(Z(I)+Z(I+1)-2*M(I+2))*A*A*A/(B*B)
1470 RETURN
1480 :
1490 :
1500 REM **** SET AKIMA ****
1510 REM CALCUALATE AKIMA COEFS
1520 FOR I=1 TO V-1
1530 REM SHIFT I TO I+2
1540 M(I+2)=(Y(I+1)-Y(I))/(X(I+1)-X(I))
1550 NEXT I
1560 M(V+2)=2*M(V+1)-M(V)
1570 M(V+3)=2*M(V+2)-M(V+1)
1580 M(2)=2*M(3)-M(4)
1590 M(1)=2*M(2)-M(3)
1600 FOR I=1 TO V
1610 A=ABS(M(I+3)-M(I+2))
1620 B=ABS(M(I+1)-M(I))
1630 IF A+B<>0 THEN Z(I)=(A*M(I+1)+B*M(I+2))/(A+B)
1640 IF A+B=0 THEN Z(I)=(M(I+2)+M(I+1))/2
1650 NEXT I
1660 RETURN
```

Commodore 64:

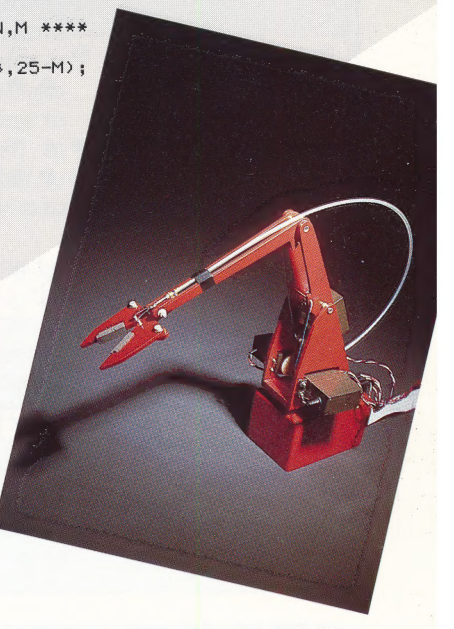
```
10 REM ************************
20 REM ** CBM CUBIC SPLINES **
30 REM ************************
40 :
50 V=7:FOR I=1 TO 25:DW$=DW$+CHR$(17):NEXT
60 DIM X(V+3),Y(V+3),M(V+3),Z(V+3)
70 X(2)=0:X(3)=5:X(4)=20:X(5)=25
80 Y(2)=2:Y(3)=12:Y(4)=12:Y(5)=2
90 REM ** LINEARLY EXTEND THE ENDS **
100 X(1)=X(2)-(X(3)-X(2))
110 Y(1)=Y(2)-(Y(3)-Y(2))
120 X(6)=X(5)+(X(5)-X(4))
130 X(7)=X(6)+(X(6)-X(5))
140 Y(6)=Y(5)+(Y(5)-Y(4))
150 Y(7)=Y(6)+(Y(6)-Y(5))
160 PRINTCHR$(147)
170 PRINT"DEMONSTRATION OF A ROUTINE WHICH"
180 PRINT"INCORPORATES A CUBIC SPLINE"
190 PRINT"JOINING KNOTS"
200 GOSUB 3000:REM DRAW KNOTS
230 GOSUB 1000:REM SET AKIMA
235 J=2
240 FOR X=X(2) TO X(5)
250 GOSUB2000:REM AKIMA FUNCTION
255 IF X=X(J) THEN J=J+1:GOTO270:REM DON'T DRAW
260 N=X:M=Y:GOSUB5000:PRINT CHR$(46)
270 NEXT
280 GET A$:IF A$="" THEN 280
290 END
1000 REM **** SET AKIMA ****
1010 REM CALC AKIMA COEFS
1020 FOR I=1 TO V-1
1030 M(I+2)=(Y(I+1)-Y(I))/(X(I+1)-X(I))
1040 NEXT I
1050 M(V+2)=2*M(V+1)-M(V)
1060 M(V+3)=2*M(V+2)-M(V+1)
1070 M(2)=2*M(3)-M(4)
1080 M(1)=2*M(2)-M(3)
1090 FOR I=1 TO V
1100 A=ABS(M(I+3)-M(I+2))
1110 B=ABS(M(I+1)-M(1))
1120 IF A+B<>0 THEN Z(I)=(A*M(I+1)+B*M(I+2))/(A+B)
1130 IF A+B=0 THEN Z(I)=(M(I+2)+M(I+1))/2
1140 NEXT I
1150 RETURN
2000 REM **** AKIMA FUNCTION ****
2010 N=1
2020 IF X<X(1) OR X>X(V-2) THEN RETURN:REM CHECK RANGE
2030 I=0
2040 I=I+1:IF X>=X(I) THEN 2040
2050 I=I-1
2060 REM BEGIN INTERPOLATION
2070 B=X(I+1)-X(I)
2080 A=X-X(I)
2090 Y=Y(I)+Z(I)*A+(3*M(I+2)-2*Z(I)-Z(I+1))*A*A/B
2100 Y=Y+(Z(I)+Z(I+1)-2*M(I+2))*A*A*A/(B*B)
2110 RETURN
3000 REM **** PRINT KNOTS ****
3010 FOR I=2 TO 5
3020 N=X(I):M=Y(I):GOSUB 5000:PRINT CHR$(215)
3030 NEXT I
3040 RETURN
5000 REM **** POSITION AT N,M ****
5010 PRINT CHR$(19);
5020 PRINT TAB(N);LEFT$(DW$,25-M);
5030 RETURN
```

Basic Flavours

Spectrum:

Delete lines 1040 and 1050 from the BBC Micro version and insert the following:

```
1170 CLS
1220 CIRCLE X(I)*5-4,Y(I)*5+15,10
1270 IF X=X(2) THEN PLOT X*5,Y*5
1280 IF X<>X(2) THEN DRAW X*5,Y*5
```



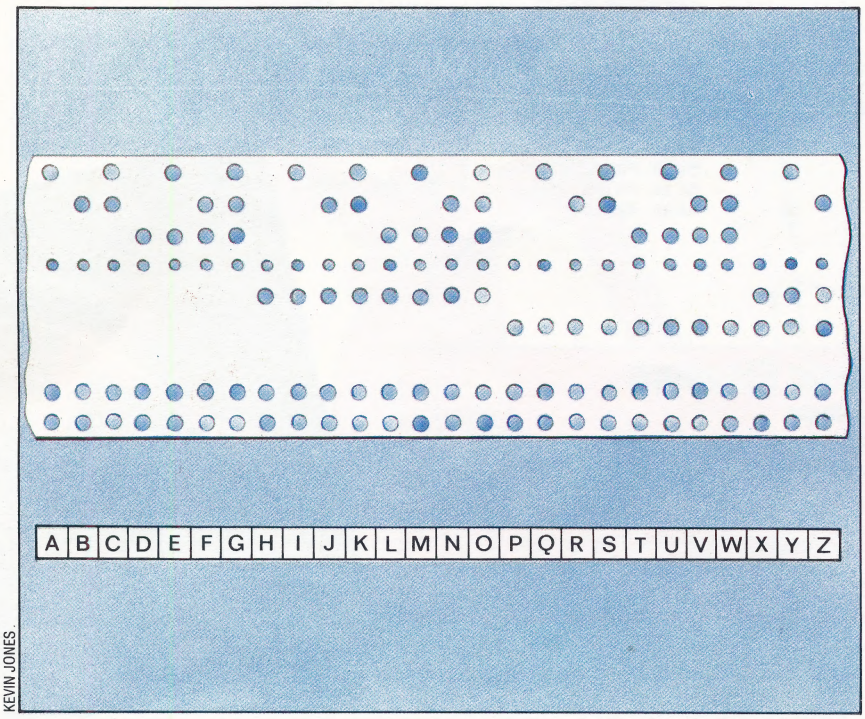
PAPER TAPE

Now largely obsolete, *paper tape* was for many years the sole method of inputting programs and data into a computer. The system worked rather like a telex or teleprinter machine (from where, in fact, the system originated). A typist would enter the program from a keyboard that punched holes into a long strip of paper held on a continuous reel. Once completed, the tape would be fed through a reading device that mechanically read the paper and converted the perforated patterns into electronic ASCII code. So, instead of programs, operating systems and languages were stored on paper tape that had to be fed through the computer to 'boot' the system.

The tape (about 2cm wide) can accommodate five, seven or eight positions (or tracks) on which the holes could be punched. Down the centre of the tape is a line of smaller holes, originally used as the mechanism by which paper was fed into the reader. Later, paper tape machines used faster methods of feeding paper into the machine; however, the line of feed holes has been retained as many of the synchronisation processes used by the readers have come to depend on it. Modern paper tape readers use much faster optical reading methods. This means that the paper used on modern reels must not be transparent to the optical readers.

Although there are no home computers that incorporate paper tape, modern micros still retain a legacy from the days of computer tape; the eight-bit character set used by paper tape still forms the basis of the electronic ASCII codes used by microcomputers. There are also many older mainframes that use paper tape for a number of applications, although — like microcomputers — there are no facilities to accommodate paper tape being used on the modern machines.

All Taped Up
Here we see the ASCII alphabet as represented on paper tape. The bits are arranged from left to right in descending order, with the paper feed line (the smaller holes) between bits three and four. Bits 6,7 and 8 are used to provide shift and control functions. Bit 8 is the parity line, which provides a check whether the code transmitted has been received correctly



KEVIN JONES

PARALLEL INPUT/OUTPUT

Parallel input/output (PIO) is a feature of some interfaces between a computer and its associated peripherals. Data is transmitted to the peripheral and vice versa in 'parallel' — namely, for an eight-bit microcomputer, each of the bits within a byte will be transmitted simultaneously down eight separate lines of the interface. Similarly, the peripheral will transmit eight-bit data patterns back to the computer in the same way, although it should be noted that these will not necessarily be sent along the same lines.

The advantage of this system is that the speed of transmission is much faster than if information is sent and received down a single line — known as 'serial input/output' (SIO). The disadvantage of parallel methods of transmission is the extra cost involved. Instead of a single cable being built into the system, eight separate lines must be fitted.

Although there are few functions in which parallel input/output is of vital importance, the system tends to be used when transmission speed is important. The most common use of PIO on home computers can be seen in the expansion or cartridge bus. Here, the computer will be communicating with another eight-bit chip and, to acquire the maximum efficiency, it is very important that data be transferred between them as quickly as possible. Another common use of the parallel input/output system is evident in a parallel printer connection, in which each character to be printed is sent as an eight-bit ASCII code. If the data is sent in a serial fashion, the printer will have to wait until all the bits have arrived before it can print the character, which will slow the print rate. However, in a PIO system, all the bits of the character arrive together and can be stored in the printer's buffer, allowing the printing speed to be increased.

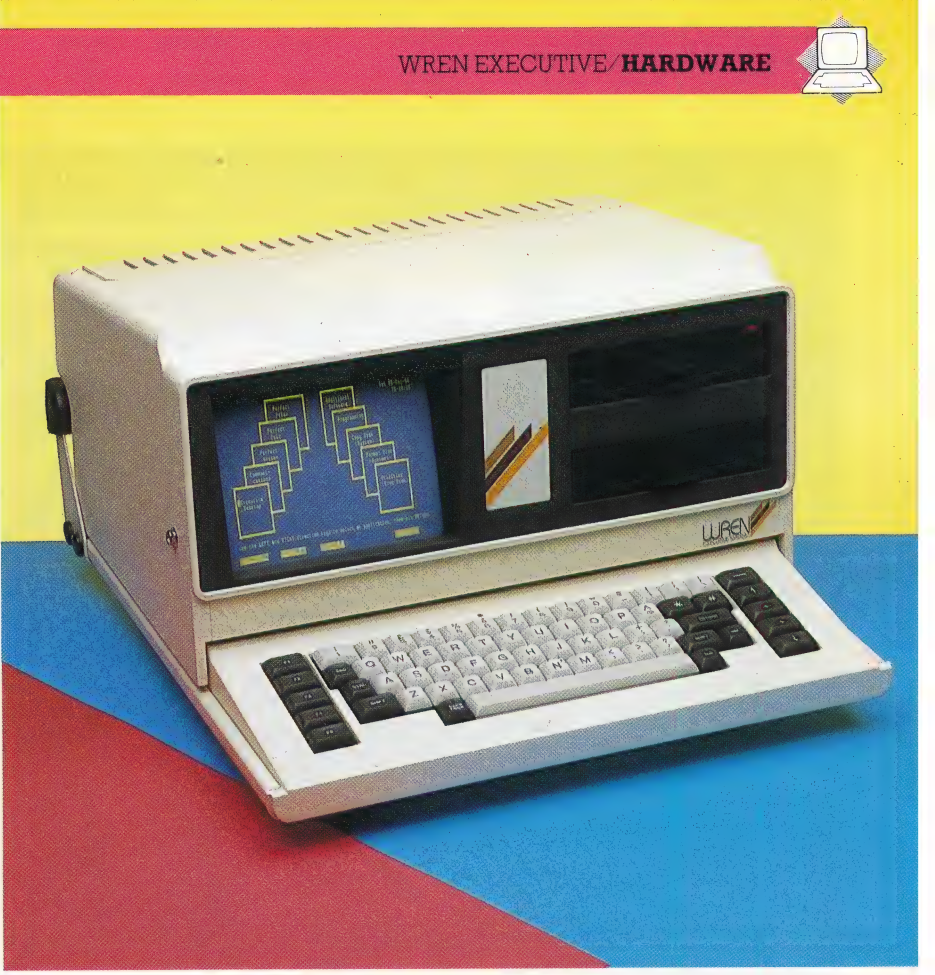
PARALLEL PROCESSING

In its strictest sense, *parallel processing* refers to several processes being executed in parallel. The implication of this is that the computer should have more than one processor, as a single processor can only execute one instruction at a time, regardless of its speed.

However, the term *parallel processing* — especially in advertisements for computers — is now used to imply the ability of the computer to work on several different programs at once. For example, a user could be running one program to calculate business accounts while the computer is running a second program at the same time, feeding information to a printer. Of course, these programs are not being executed by the processor simultaneously. What is happening is that the computer will run the accounts program until it recieves a message from the printer that the buffer is empty. The processor will then interrupt the accounts program, refill the buffer and then continue with the accounts. Providing the action is fast enough, you will not notice any interruption and the programs will *appear* to run in parallel.

FINDING ITS NICHE

As a portable, self-contained unit with two disk drives, a built-in auto-dial modem and BBC BASIC, the Wren Executive computer is certainly not lacking in capabilities. But a price of under £1,000 and an eight-bit processor place it between the business and home micro markets. We examine its chances of finding a widespread popularity.



CHRIS STEVENS

An Unusual Bird
Distributed by Opus Supplies, the Wren Executive computer is an inexpensive portable machine, which has the unusual feature of running both BBC BASIC and the CP/M operating system. Other features include twin disk drives, monochrome monitor and a built-in modem

The Wren Executive computer has already had a chequered history. The machine, built by Thorn EMI, was originally marketed by Prism. When that company went into liquidation at the beginning of 1985, it seemed that the Wren would be yet another victim of the turbulent computer industry. Recently, however, distribution of the computer was taken over by Opus Supplies, which is planning to relaunch the machine. In this article we assess the machine's chances of acquiring a niche in the highly competitive business market.

The Wren Executive is designed to be a 'portable' machine. That is to say, it is a self-contained package consisting of computer, monitor and disk drives. Although the machine appears at first sight to be encased in painted metal, the casing is in fact made almost entirely of textured plastic, with only the base and the keyboard surrounds constructed of metal. At the rear of the machine is a carrying handle. Like many portable machines, the computer is perhaps a little too heavy for long periods of carrying, and is perhaps better suited as a desk top computer that can be transported from one work area to another.

The keyboard has the usual QWERTY configuration, with the only difference being that the * and # symbols have their own keys. To the left of the typewriter keys are five programmable function keys, which when used in conjunction with either the Control or Shift keys, can provide up to 15 different functions. These keys have different purposes depending on which application is being used, but they are predominantly used for single keyword entries of CP/M commands such as PIP and RENAME. On the opposite side of the keyboard are the four cursor keys and a Home key to return the cursor to the top left-hand corner of the screen.

The screen measures 150 by 105mm and is typical of monochrome monitors provided with business machines, with a text resolution of 80 by 25 characters. The only noticeable difference is that the foreground colour is orange as opposed to the usual green. This makes for a bright and attractive display and should be suitable for long periods of work. To the right of the monitor screen are a pair of 5¼in floppy disk drives.

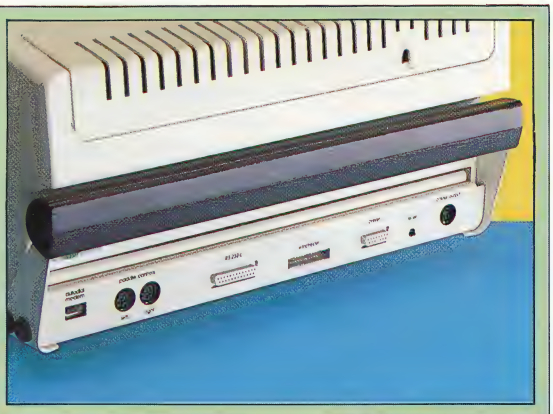
One problem with the screen is the design of the casing. The Wren sits fairly low on the desk with no legs provided to tilt the machine with the screen. The screen is embedded back into the casing with a 40mm ledge projecting out above the screen. Although this has been angled, you may find that the ledge obscures the top of the screen if you are rather tall or usually sit close to the keyboard. This is not a serious difficulty, but it is slightly annoying.

The Wren is well supplied with interfaces. There are a number of ports to allow the computer to be connected to a wide range of peripherals at the rear of the machine. And unlike many similarly priced business machines, the Wren has a built-in autodial modem. The Wren owner is therefore able to access all the major databases such as Prestel and Micronet without having to purchase any other equipment (although, of course, you will have to become a subscriber to one of the databases before they will allow you to log on).

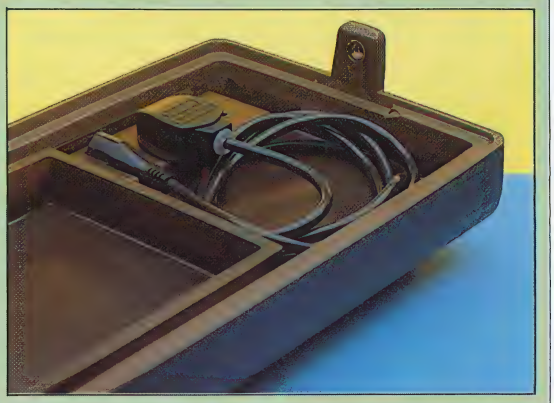
The Wren even comes with a cable to connect it to the telephone system, which slots into the British Telecom-standard 600-series socket on the rear of the computer. The communications software provided with the Wren allows a number of different telephone numbers to be stored that can be autodialled when required.

Next to the communications socket is a pair of paddle ports that are provided to accommodate the fitting of external control devices, such as a mouse and joysticks. Beyond these is a 25-way D connector providing the interface for RS232C

A Wealth Of Interfaces
One of the best features of the Wren is the wealth of interfaces fitted to the machine as standard. These include not only the conventional Centronics and RS232C interfaces, but also a BT–series 600 telephone jack for the modem (a cable is included). There is also a parallel drive to enable a Winchester hard disk to be fitted



Plug In And Go
The monitor and disk drives are protected in transit by a heavy duty plastic cover, which fits over the front of the machine. In keeping with the philosophy of being able to 'plug in and go', the manufacturers have taken the unusual step of providing a power cable with the plug already fitted. The power cable is stored in the casing during transportation



Words Of Perfection
As the Wren is built by Thorn EMI, it seems natural that they should bundle their own series of software applications with the machine – Perfect Writer, Perfect Filer and Perfect Calc. Also included is a general program called Executive Desktop, communications software and BBC BASIC

serial communications. RS232 sockets are normally used on business computers for external modems. This is obviously not a requirement on the Wren and so will primarily be used for local area networks and serial printers.

On the left of the RS232 port is a Winchester parallel interface into which a hard disk storage system can be fitted. The addition of a Winchester disk system can add anything from 500 Kbytes to 50 megabytes of additional storage to the system. There is also a Centronics-compatible parallel interface that provides the outlet for hard copy to be printed. Finally, on the far right, is an RGB colour monitor socket. Between the printer and monitor interfaces is a tiny reset button that will provide a cold start to the system.

The Wren is based around the familiar Z80B eight-bit processor. This is perhaps the computer's greatest weakness. At a time when home micro manufacturers are frantically attempting to compete with each other in the development of 16-bit micros, the Wren appears to look somewhat

Monitor
The Wren is fitted with a built-in monochrome screen that is bright and easy to read

RS232C Port
Direct communication with other computers and other serial applications is provided through this socket

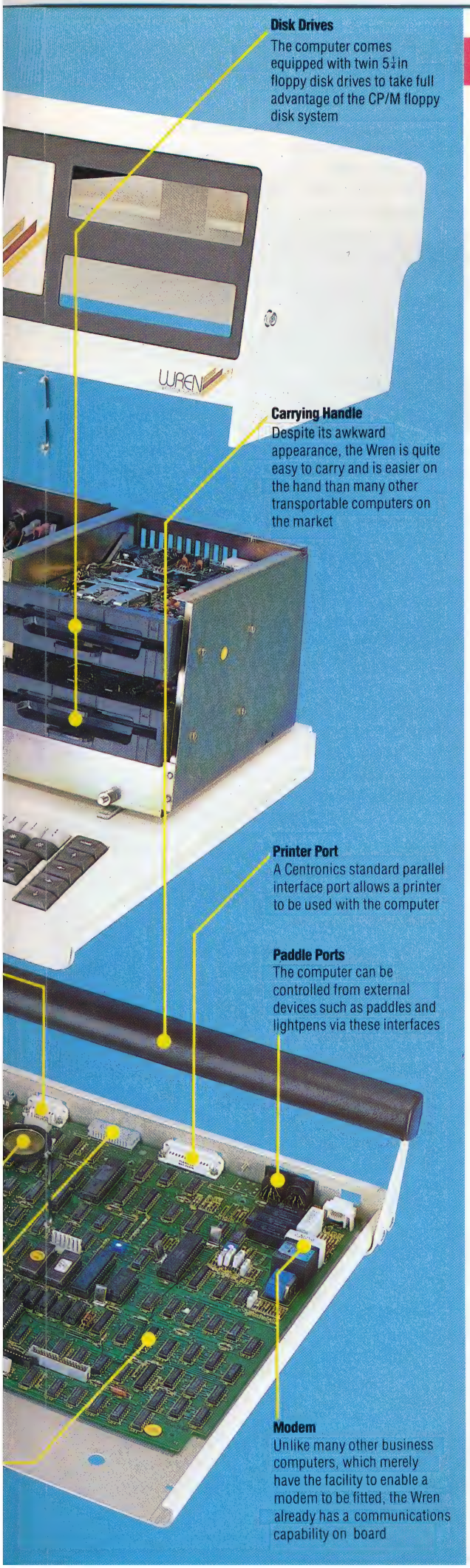
Colour Monitor Socket
Should the user wish to take advantage of the Wren's colour facilities, the computer has an additional RGB monitor port, which fits to an external monitor

Speaker
Sound facilities are provided by the Wren's own on-board speaker

Winchester Interface
A Winchester hard disk, which greatly increases the storage and retrieval capacity of the computer, can be plugged into this parallel interface

RAM Chips
Despite having only an eight-bit processor, the Wren is able to address up to 400 Kbytes of RAM through the process of bank switching

CHRIS STEVENS


Disk Drives
The computer comes equipped with twin 5¼in floppy disk drives to take full advantage of the CP/M floppy disk system
WREN
Carrying Handle
Despite its awkward appearance, the Wren is quite easy to carry and is easier on the hand than many other transportable computers on the market
Printer Port
A Centronics standard parallel interface port allows a printer to be used with the computer
Paddle Ports
The computer can be controlled from external devices such as paddles and lightpens via these interfaces
Modem
Unlike many other business computers, which merely have the facility to enable a modem to be fitted, the Wren already has a communications capability on board

antiquated in its processing. Although reasonably fast by eight-bit standards, it cannot be compared with the speed of 16-bit micros such as the IBM PC or the Apple Macintosh.

As the Wren is based around the Z80 processor, it was natural that the designers chose to use CP/M as the operating system. The trouble here is that by recent standards CP/M is a rather unfriendly operating system despite its power. The designers have alleviated this problem by incorporating a menu-driven procedure into the operating system allowing you to choose an application by means of a cursor and pressing RETURN. The system will then prompt you to insert the appropriate disk and will run the application automatically. Thus many users will find that they have no need to use CP/M directly as almost everything the average business user needs is provided within the menu.

The applications disks bundled with the computer are the CP/M system disk, Perfect Writer (a word processor), Executive Desktop (a package to keep time, dates, diary and appointments), the Perfect Filer database, the Perfect Calc spreadsheet and a disk containing the communications software and BASIC.

The BASIC used in programming the Wren surprisingly turns out to be standard BBC BASIC. Although the machine has almost a full implementation of the language, being based around a Z80 processor instead of a 6502 has led to some differences, mostly in the input/output routines. The implementation of the language also appears to have generated some bugs. For example, in modes 0 and 1, the cursor disappears from view. The Delete key does not work in these modes, and backtracking with the cursor and attempting to overwrite merely results in the new character being superimposed onto the old one. This prevents any serious programming in these modes, although the others seem to work perfectly well.

Ultimately, the future success of the Wren Executive depends on whether it can find itself a suitable niche in the market. With a built-in modem, CP/M operating system and BBC BASIC at under £1,000, the Wren seemed exceptional value when it was launched in the early part of 1984. Since then the sales potential of eight-bit business machines, even at the low end of the market, has been rapidly diminishing. Yet the machine is still far too expensive to make a serious attack on the home market. This is especially true now that Amstrad has implemented CP/M in its range of home micros.

This leaves the educational market in which, it seems, the Wren may make its biggest impact. The fact that the machine uses BBC BASIC, which is widely used in schools, coupled with a built-in modem to take advantage of the various schools databases that are becoming available, could make the machine attractive to educational establishments attempting to get the maximum value for their money.

WREN EXECUTIVE

PRICE
200 Kbyte version: £1,138; 400 Kbyte version: £1,493 (both inc VAT)

DIMENSIONS
440x410x250mm

CPU
Z80B, running at 6 MHz

SCREEN
80x25 text screen, 640x256 graphics screen

INTERFACES
Monitor port, Centronics interface, RS232C port, BT series-600 telephone jack, twin paddle ports, Winchester disk interface

LANGUAGES AVAILABLE
BBC BASIC

KEYBOARD
57 typewriter keys, five programmable function keys, five cursor control keys

DOCUMENTATION
Apart from the User manual, each of the applications provided has its own manual. The documentation is well laid out and provides the user with full explanations

STRENGTHS
The number of interfaces, together with a built-in modem and the wide number of bundled programs make the machine excellent value

WEAKNESSES
The 8-bit technology used by the Wren is becoming increasingly dated. There also appear to be a few bugs in some of the applications

MUTINY ON THE BOUNTY

In addition to the major and minor events that can affect the progress of the voyage, there is also the chance the crew will mutiny and end the game. Several factors can instigate the revolt, each lowering the threshold of the crew's tolerance, leading to a battle in which the insurgents take over the ship and sail back home.

There are several major and minor contingencies that affect the journey in positive or negative ways. These can be selected at random during each week and executed, but will not occur again once they have been selected. There is one more major event that will arise at any time during the game if certain criteria are met. If conditions on the ship become intolerable, the crew will mutiny.

There are eight factors that can precipitate a mutiny. Although the ship has room for a crew of 16, any number over 12 is considered overcrowded and the cramped conditions will make the crew unhappy. If a cook was not hired, or dies during the voyage, the crew will have to perform kitchen duties, which will lessen the quality of the food, and create more unrest!

Because sighting an albatross is a good omen (see page 1172), the crew will be relatively contented; but if they kill it, which brings bad luck, a mutiny will become more likely. Being put on half rations of any provision will further promote unrest in the crew, as will concern over the ship's kitty becoming smaller than the wages owed. Finally, the crew were informed when they were hired that the voyage would last eight weeks. Any time over this figure will create grumblings among them, which will intensify with each extra week needed to complete the expedition.

CONDITION TESTING

To test if the conditions are sufficient to start a rebellion, a mutiny factor, MF, is created. Each condition is tested at the start of a week by calling a mutiny routine from the main program loop. If the result of each test is positive, a value is added to the mutiny factor. This procedure continues until MF reaches 100, at which time the mutiny will take place. There is also a random factor, of up to 30, which is included in the weekly assessment of the mutiny factor.

Line 879 sends the main program to the subroutine at line 7200, which makes a weekly check of the mutiny factor MF, which is set to 0 at the beginning of the routine. If the crew are put on half rations at any stage of the voyage, the variable H$ will be set to Y. Because getting enough food is such an important consideration to the crew, giving them half rations of any provision will set H$

Module Ten: A Mutiny

Mutiny Routine

```
879 GOSUB7200
```

Addition To Main Voyage Loop

```
7200 REM MUTINY
7210 MF=0
7215 IFH$="Y"THENMF=MF+30
7220 NC=0
7225 FORT=1TO16
7228 IFTS(T,1)=5ANDTS(T,2)<>0ANDTS(T,2)<>-999THENN
C=1:T=16
7230 NEXT
7235 IFNC=0THENMF=MF+30
7240 IFA$="Y"THENMF=MF-20
7245 IFB$="Y"THENMF=MF+30
7250 IFCN>12THENMF=MF+30
7255 IFWT>MOTHENMF=MF+30
7260 IFWK>8THENMF=MF+((WK-8)*10)
7275 MF=MF+INT(RND(1)*30)
7280 IFMF<75 THENRETURN
7282 PRINTCHR$(147)
7284 IFMF>100THEN7300
7285 S$="CONDITIONS ON THE SHIP*":GOSUB9100
7286 S$="ARE GETTING WORSE*":GOSUB9100
7287 S$="AND SOME OF THE CREW*":GOSUB9100
7288 S$="ARE TALKING OF MUTINY!*":GOSUB9100
7290 PRINT:GOSUB9200
7292 S$=K$:GOSUB9100
7294 GETI$:IFI$=""THEN7294
7299 RETURN
7300 PRINTCHR$(147)
7305 PRINT:GOSUB9200
7310 S$="THE CREW HAVE MUTINIED*":GOSUB9100
7312 S$="BECAUSE:*":GOSUB9100
7313 X=0
7314 IFH$<>"Y"THEN7320
7315 GOSUB9200:X=X+1:PRINTX;
7316 S$="THEY HAVE BEEN ON 1/2 RATIONS*":GOSUB9100
7318 S$="   FOR SOME OF THE VOYAGE*":GOSUB9100
7320 IFNC<>0THEN7325
7321 GOSUB9200:X=X+1:PRINTX;
7322 S$="THERE IS NO COOK*":GOSUB9100
7324 S$="   AND THE FOOD IS AWFUL*":GOSUB9100
7325 IFB$<>"Y"THEN7330
7326 GOSUB9200:X=X+1:PRINTX;
7327 S$="THE ALBATROSS WAS KILLED!*":GOSUB9100
7330 IFCN<13THEN7335
7331 GOSUB9200:X=X+1:PRINTX;
7332 S$="THE SHIP IS OVERCROWDED*":GOSUB9100
7335 IFMO>=WT THEN7340
7336 GOSUB9200:X=X+1:PRINTX;
7337 S$="THERE ISN'T ENOUGH GOLD*":GOSUB9100
7338 S$="   FOR THEIR WAGES*":GOSUB9100
7340 IFWK<=8 THEN7350
7341 GOSUB9200:X=X+1:PRINTX;
7342 S$="THEY HAVE BEEN AT SEA*":GOSUB9100
7343 S$="   FOR MORE THAN 8 WEEKS*":GOSUB9100
7350 PRINT:GOSUB9200
7360 S$="THE CREW SEIZE THE SHIP*":GOSUB9100
7362 GOSUB9200
7363 S$="AND SAIL AWAY*":GOSUB9100
7370 GOSUB9200
7372 S$="LEAVING YOU ADRIFT IN *":GOSUB9100
7373 S$="AN OPEN BOAT*":GOSUB9100
7374 S$="LET'S HOPE YOU ARE PICKED UP":GOSUB9100
7375 PRINT:GOSUB9200
7380 PRINT"       GAME OVER"
7382 END
```

Basic Flavours

Spectrum:

Make the following changes:

```
7282 CLS
7294 LET I$=INKEY$:IF I$=""THEN GO TO 7294
7300 CLS
```

BBC Micro:

Make the following changes:

```
7282 CLS
7294 I$=GET$
7300 CLS
```

to Y for the remainder of the voyage. H$ is tested by the mutiny routine, adding 30 to MF if H$ holds Y. Line 7215 checks H$ and increments MF if necessary.

The mutiny routine then checks whether there is a cook on board by setting the variable NC to 0, and sets up a loop from 1 to 16, going through the first element of the crew strength/type array TS(,) for a 5, representing a cook. If the cook's strength is not 0 or -999, implying a healthy cook, NC is set to 1. If no cook was hired, or he has died during the voyage, NC remains at 0. Line 7235 tests NC and adds 30 to the mutiny factor if NC equals 0.

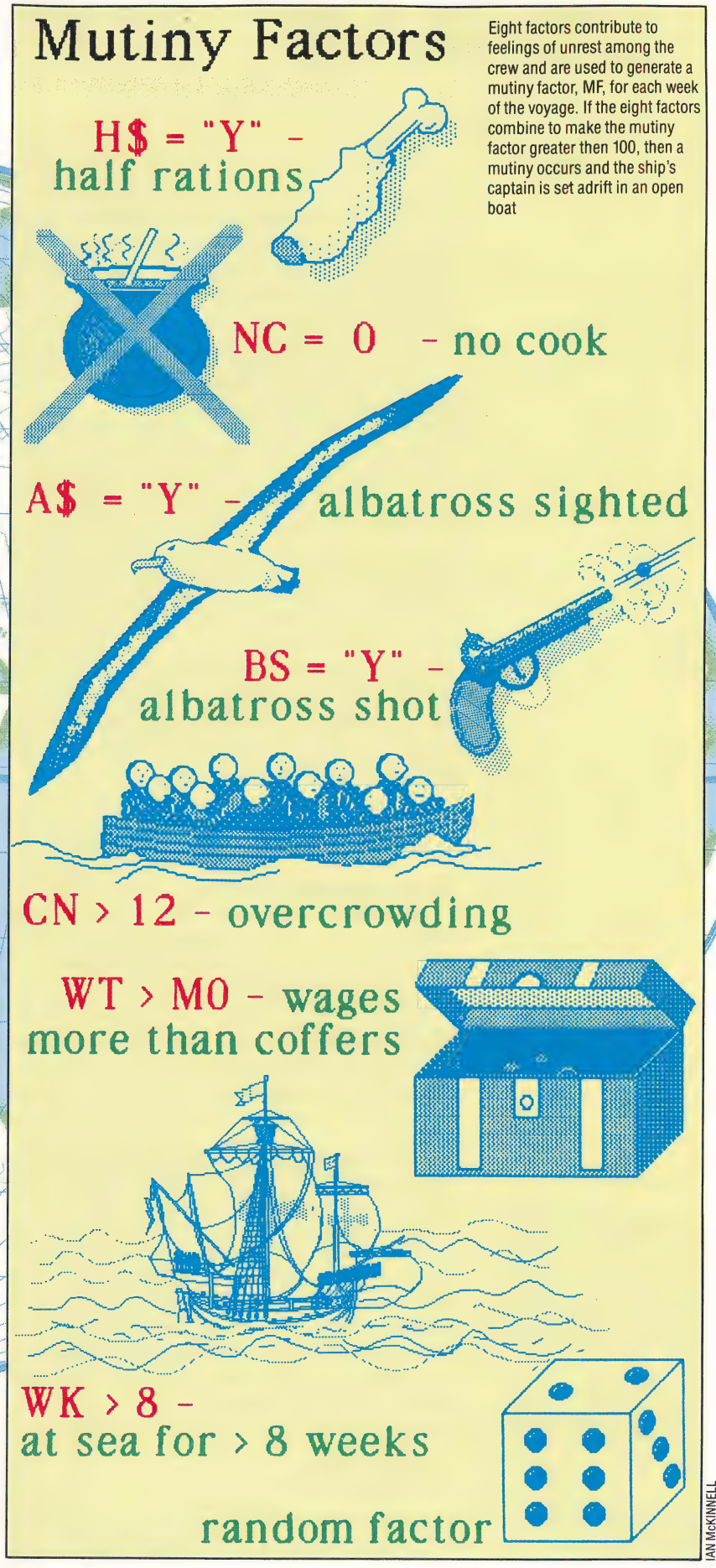
If the albatross has already been sighted during the voyage, then A$ is set to Y at line 6055 of the albatross subroutine. This brings good luck by removing 20 from the mutiny factor at line 7240. If you have shot the albatross, line 6162 sets B$ to Y, which brings bad luck, and line 7245 increases the mutiny factor by 30. Line 7250 checks whether the number of crew is greater then 12 and, if so, adds 30 to the mutiny factor, thus registering the crew's general feelings about the overcrowded conditions.

The total wage bill is represented by WT and the money remaining in the captain's coffers by MO. Line 7255 checks if the wage bill is greater than the money in the kitty and, if so, adds another 30 to MF. The mutiny factor is increased by 10 for each additional week of the journey after eight weeks. Line 7260 checks whether the journey has in fact taken over eight weeks and, if this is the case, the equation subtracts 8 from the number of weeks (WK), multiplies the remainder by 10 and adds this result to MF. On line 7275, a random factor between 0 and 29 is added to MF.

If, after checking all the conditions, the total for the mutiny factor is less than 74, control is returned to the main program by line 7280. If the factor is greater than 100, line 7284 sends the program to 7300 — at which point the mutiny takes place. If the factor is between 75 and 100, lines 7285 to 7288 warn you that conditions on the ship are getting worse and the crew are talking of mutiny, before returning control to the main program.

If the crew does mutiny, the program determines the causes and prints the reason. Line 7314 checks whether H$ is equal to Y and informs you that the crew have been on half rations for part of the voyage. Line 7320 checks whether NC is equal to 0, signifying there is no cook on board, and you will be told that the mutiny was a result of the lack of culinary expertise. Line 7325 determines if the albatross was killed or not and informs you that, if the bird was killed, it was a contributing factor to the mutiny. Line 7330 checks whether the ship was overcrowded, 7335 determines if there is enough money to pay the crew and 7340 looks at the journey length.

Finally, you will be told that many of the crew have been killed and that the insurgents have seized control of the ship and are sailing away. Because the mutiny signifies the end of the game, it is important, if you are to be successful, to take good care of your crew.


Mutiny Factors
Eight factors contribute to feelings of unrest among the crew and are used to generate a mutiny factor, MF, for each week of the voyage. If the eight factors combine to make the mutiny factor greater then 100, then a mutiny occurs and the ship's captain is set adrift in an open boat
H$ = "Y" - half rations
NC = 0 - no cook
A$ = "Y" - albatross sighted
BS = "Y" - albatross shot
CN > 12 - overcrowding
WT > MO - wages more than coffers
WK > 8 - at sea for > 8 weeks
random factor

IAN McKINNELL

DYNAMIC DUO

In this penultimate instalment in our series on PASCAL, we break away from the limitations thus far imposed on our files by implementing dynamic allocation. We look at concepts such as de-allocation, pointers, heaps and linked structures, which provide us with the freedom necessary to process files of unspecified size.

PASCAL provides two standard procedures (new and dispose) and a special data type, which together offer extraordinary power and flexibility in the 'dynamic allocation' and de-allocation of memory. A 'pointer' type is simply a variable which, instead of containing a data value such as an integer, is a 'pointer to' an integer or any other data object, structured or otherwise. The notation used is similar to that when we denote a file look-ahead buffer (which effectively points to the next record on the file, though in a totally different way). A pointer type is defined by preceding the identifier (of the data type we wish to point to) with an up-arrow:

```
TYPE
   LongArray   = ARRAY  1 . . 100  OF real;
   indirect    = ↑LongArray;
```

The same alternative symbol ('@') is supported by all ISO PASCAL compilers. The type definition reserves a single memory location that will be used to hold the address of a large array only when it should be created with the procedure new. In the meantime, the pointer's value is undefined, just as any other variable would be, so that:

```
VAR
   address     : indirect;
   number      : integer;
```

would reserve space for one machine address (16 bits on an eight-bit micro) and one integer, with neither being initialised to any particular value.

Just as we might want to initialise the integer to zero before use, the pointer may be specifically assigned the special value NIL. This is a reserved word in PASCAL, merely signifying that the pointer doesn't point to anywhere useful, and is the equivalent of the numeric value zero (signifying the absence of any actual number). The two variables in the example could therefore be assigned thus:

```
address := NIL
number :=0;
```

As NIL is a constant value belonging to a generic type, it would perhaps be preferable to define it as an identifier in a language. Wirth obviously changed his mind on this aspect because in MODULA-2 the same word is in fact a predefined identifier, not a reserved word. In order to demonstrate the use of new and dispose we'll just use integers for simplicity.

When a pointer variable is needed to point to a new item, we call the PASCAL procedure new, which allocates space for the item and puts its address into the pointer. The data item is then referred to by 'de-referencing' the pointer, p, with the notation p↑. Notice that the up-arrow now follows the pointer identifier, as with file–buffer notation, and may be thought of as 'the item that p is pointing to'. Needless to say, it is an error to de-reference a pointer that is undefined or one that points nowhere.

DYNAMIC DE-ALLOCATION

When we have finished with any data created by newing pointers, PASCAL's dispose procedure may be called. This is the opposite of new, in that the memory is returned to the dynamic pool and the pointer value becomes undefined. The program TwoPlusTwo illustrates these points:

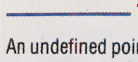
```
PROGRAM TwoPlusTwo (output);

TYPE
      pointer    = ↑ integer;

VAR
      p1,
      p2         : pointer;
      answer     : integer;

BEGIN
   new (p1);
   p1 ↑ :=2;
   new (p2);
   p2 ↑ := p1 ↑;
   answer := p1 ↑ + p2 ↑;
   WriteLn (p1 ↑,'+', p2 ↑, '=', answer );
   dispose (p1);
   dispose (p2);
END.
```

This extremely peculiar method of adding two and two illustrates two important points:

- Dynamic variables are anonymous — there is no variable identifier (like N) that holds the value 2 anywhere in the program; these items are referred to indirectly.
- After the second dispose the only memory used is the single integer (answer); all the dynamic data no longer exists.

If you have followed our Assembly language series, you will have already appreciated that the indirection implicit in the use of pointers is analogous to indirect addressing at machine level. There are, however, big differences in the use of indirection in such a high-level language as PASCAL. First, we never know (or need to know) what the actual addresses are. The only 'absolute address' available to the PASCAL programmer is NIL.

Pointer Symbols



An undefined pointer (after a declaration, before an assignment or after dispose)

A pointer pointing 'nowhere' (after an assignment to NIL)

A record node with one data field and one pointer

Also, we are free to use, reclaim and subsequently re-use any of the memory available without the need to organise 'garbage collection' ourselves. The memory management is taken care of by PASCAL, and the only information we may require is how much memory is left. This is obtained by the non-standard function MemAvail or, in some implementations such as Acorn ISO PASCAL, the function Free. This returns the number of bytes remaining in memory.

Another useful extension is the function SizeOf (TypeIdentifier) which returns the size in bytes of any particular type. The user RAM is divided into two internal structures — the stack and the 'heap'.

The stack is used for procedure and function calls, storing their return addresses, local data and returned values. All dynamic data is allocated on the heap. This operates in a similar way to the stack except that it is not a LIFO (Last In First Out, see page 948) data structure and the heap starts at the opposite end of user memory, growing towards the stack. With over-ambitious use of new and/or recursion, a 'stack-heap collision' could occur, but this can be avoided with:

```
IF SizeOf (thing) > PerCent * MemAvail THEN . .
```

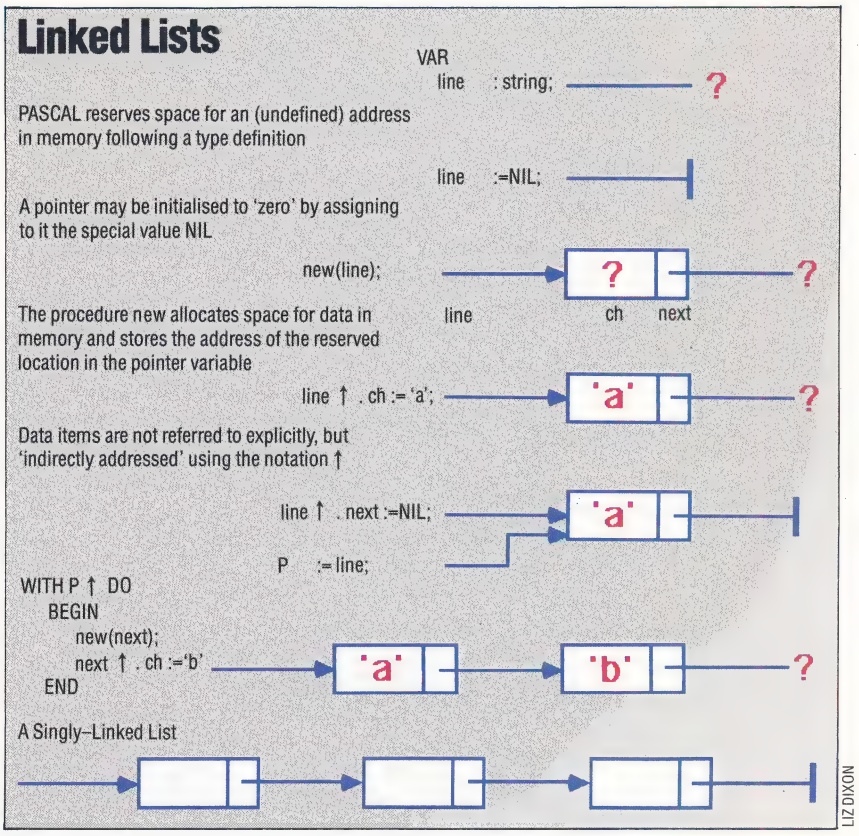
where thing is the type identifier of the data about to be created and PerCent is a value in the region of 0.7, allowing 30 per cent of the memory for the stack and 70 per cent for the heap. If, as is usually the case, MemAvail/Free operates as a 'high water mark', an additional 'garbage count' of disposed items may be maintained.

LINKED STRUCTURES

The real power of pointers is realised when we create linked structures such as trees, singly- or doubly-linked lists, circular structures and so on. If we consider the problem of character strings, we can, and often do, use an array of some fixed size, say 80 characters. If we have an array of strings to store a document, for instance, each blank line will still take up 80 bytes of storage. Equally, we can't represent lines longer than 80 characters — the whole data structure is simply too rigid.

A PASCAL compiler has to distinguish between identifiers of any length. So, for example, how can we accurately model this real-world characteristic of varying size? The natural structure to use would be a record containing two fields : a field for each data item (chars in this case) and a second pointer field which points to the next record, if any, in the list.

```
TYPE
     string     = ↑character;
     character  = RECORD
                    ch    : char;
                    next  : string
                    END;
VAR
     line       : string;
BEGIN
  line := NIL;
       etc.
```


Linked Lists
VAR
line : string;
PASCAL reserves space for an (undefined) address in memory following a type definition
line :=NIL;
A pointer may be initialised to 'zero' by assigning to it the special value NIL
new(line);
line
ch
next
The procedure new allocates space for data in memory and stores the address of the reserved location in the pointer variable
line ↑ . ch := 'a';
'a'
Data items are not referred to explicitly, but 'indirectly addressed' using the notation ↑
line ↑ . next :=NIL;
P := line;
WITH P ↑ DO
BEGIN
new(next);
next ↑ . ch :='b'
END
'a'
'b'
A Singly-Linked List

An empty string is represented simply by assigning the value NIL to the string. Any other sequence of characters will require a new record to contain each char and another pointer to point to the next record. The last record will have its next field initialised to NIL so that the end of the string may be detected. A procedure to print the string would then be:

```
WHILE line <>NIL DO
  BEGIN
    write (line ↑. ch);
    line := line ↑. next
  END
```

Notice that the data structure is a recursive one as it is defined in terms of itself. The pointer field cannot be given a type that has been fully defined, so a 'forward' reference is allowed.

In Succession
PASCAL ofers the programmer the possibility of implementing 'linked lists' — powerful data structures in which each item points to the next item on the list. Lists may be singly-linked (between successive items), doubly-linked (i.e. each element has pointers to both the next and last item in the list), or circular (where the last item in a singly-linked list points back to the first item in the list)

Circular List Program

```
PROGRAM   CircList  ( input, output );
{_____________________________________________
___ Intent : TO insert records containing
___  mixed data into a dynamically allocated
___  circular list. Data is placed IN order
___  OF a key alphabetic field (Name) thus
___  sorting algorithms are unnecessary.
_____________________________________________}

CONST
      StringLength =  25;
      space        = ' ';

TYPE
      Cardinal     = 0 .. MaxInt;
      StringSize   = 1 .. StringLength;
      string       = PACKED ARRAY [ StringSize ]
                           OF char;
      thing        = RECORD
                       Name     : string;
                       { . . other fields }
                       debt     : Cardinal
                     END; { thing }

      pointer      = ↑ node;
```

The Circle Line
This program allows you to insert records containing mixed data into a dynamically allocated circular list. Data is placed in order of a key alphabetic field (Name), and therefore sorting alforithms are unnecessary

```
      node          = RECORD
                        item    : thing;
                        next    : pointer
                      END; { node }
VAR
      list          : pointer;

  {1111111111111111111111111111111111111111}

PROCEDURE  ReadThings     ( head  : pointer );
  VAR
        data    : thing;
        OK      : boolean;
    {22222222222222222222222222222222}

  PROCEDURE  SkipBlanks  ( VAR  F  : text );
    { ignore leading white space }
    VAR
        done    : boolean;
    BEGIN
      done := EoF ( F );

      WHILE NOT done DO
        BEGIN
          done := ( F ↑ ) space ) OR EoLn ( F );

          IF NOT done THEN
            BEGIN
              get ( F );
              done := EoF ( F )
            END
        END

    END;  { SkipBlanks }
      {22222222222222222222222222222222}

  PROCEDURE  ReadCard   ( VAR  F  : text;
                          VAR  N  : Cardinal;
                          VAR  OK : boolean );
    { collect a non-negative number }
    TYPE
        single  = 0 .. 9;
    VAR
        digits  : SET OF char;
        done    : boolean;

        {33333333333333333333333}

    FUNCTION  Value ( digit    : char )      : single;
      { convert char TO numeric value }
      BEGIN
        Value := ord ( digit ) - ord ( '0' )

      END;    { Value }
        {33333333333333333333333}

    FUNCTION  Legal ( N          : Cardinal;
                      digit      : char )    : boolean;
      { are we about TO overflow MaxInt ? }
      BEGIN
        IF N = MaxInt DIV 10
          THEN
            Legal := Value ( digit ) <= MaxInt MOD 10
          ELSE
            Legal := N < MaxInt DIV 10

      END;    { Legal }
        {33333333333322222222222222}

  BEGIN   { ReadCard }

    digits := [ '0' .. '9' ];
    OK := NOT EoF ( F );

    IF OK THEN { found a digit ? }
      OK := F ↑ IN digits;

    N := 0;
    done := NOT OK;

    WHILE OK AND NOT done DO
      BEGIN { build N TO base 10 : }
        N := 10 * N + Value ( F ↑ );
        get ( F );
        done := EoF ( F );

        IF NOT done THEN
          done := NOT ( F ↑ IN digits );

        IF NOT done THEN
          OK := Legal ( N, F ↑ )
      END;

    IF NOT EoF ( F ) THEN
      ReadLn ( F )

  END;    { ReadCard }
    {22222222222222222222222222222222}

  PROCEDURE  ReadLine  ( VAR  S    : string );
    { read a line OF chars from the keyboard }
    VAR
        index   : 0 .. StringLength;
        symbol  : char;
    BEGIN
      SkipBlanks ( input );
      index := 0;

      WHILE NOT EoLn AND ( index < StringLength ) DO
        BEGIN { put a char into the string }
          index := succ ( index );
          read ( symbol );
          S [ index ] := symbol
        END;

      IF index < StringLength
        THEN { pad WITH NULs }
          FOR index := index + 1 TO StringLength DO
            S [ index ] := chr ( 0 )
        ELSE
          IF NOT EoLn THEN { too many chars }
            WriteLn ( 'WARNING - input truncated' );
      ReadLn

    END;  { ReadLine }
      {22222222222222222222222222222222}

  PROCEDURE  Insert  ( list        : pointer;
    VAR                data        : thing );
      scout,
      link    : pointer;
      alpha   : string;
    BEGIN { put data into the head node : }
      list ↑ item := data;
      scout := list;
      alpha := scout ↑ . next  . item . Name;

      WHILE data . Name < alpha DO
        BEGIN { walk round the list }
          scout := scout ↑  . next;
          alpha := scout ↑ . next ↑ . item . Name
        END;
            { no check FOR duplicates }
      new ( link );          { create another node }
      link ↑ . item := data; { insert the data, }
      link ↑  . next := scout ↑ . next; { & link }
      scout ↑ . next := link      { into the list }
    END;  { Insert }
      {22222222222222222211111111111111111}

  BEGIN   { ReadThings }

    write ( 'Name ? ' );
    ReadLine ( data . Name );
          { stop on a null entry : }
    WHILE data . Name [ 1 ] <> chr ( 0 ) DO
      BEGIN
        REPEAT
          REPEAT
            write ( 'Amount owing ? ' : 20 ) ;

            IF EoLn ( input ) THEN
              ReadLn ( input );

            SkipBlanks ( input )

          UNTIL NOT EoLn ( input );

          ReadCard ( input, data . debt, OK );

          IF NOT OK THEN { no digits }
            WriteLn ( '*** ERROR - ' : 20,
                'please re-enter ***' )
        UNTIL OK;

        Insert ( head, data );
        WriteLn ( '(RETURN when done)' : 40 );
        write ( 'Name ? ' );
        ReadLine ( data . Name )
      END

  END;    { ReadThings }
  {1111111111111111111111111111111111111111}

PROCEDURE Display ( list  : pointer );
  VAR
      print   : pointer;
  BEGIN { first skip the dummy head node : }
    print := list ↑ . next;
    page ( output );
    WriteLn ( 'Ordered list :' );
    WriteLn;
          { any (more) data TO print ? }
    WHILE NOT ( print = list ) DO
      BEGIN
        WITH print ↑ . item DO
          WriteLn ( debt : 8, Name : 30 );

        WriteLn;
        print := print ↑ . next
      END

  END;    { Display }
  {1111111111111111111111111111111111111111}

BEGIN     { CircList - Main PROGRAM }

  new ( list );     { create a dummy head node }
  list ↑ . next := list; { point it TO itself }
  page ( output );
  WriteLn ( 'Enter data :     surname first, then' );
  WriteLn ( 'the amount owed in (whole) pounds :' );
  WriteLn;
  ReadThings ( list );  { put data into the list }
  Display ( list );     { now print it IN order  }
  WriteLn ( '--- done ---' : 40 )

END .
```

BEAT GENERATION

The Commodore 64's dedicated sound generating chip — the Sound Interface Device (or SID chip for short) — is virtually a sound synthesiser in its own right. With sophisticated software it can even produce robotic-like speech. Here we describe ways of programming the SID chip from machine code, and suggest some applications.

Sound from the Commodore 64 is normally channelled via the RF socket straight to the television. However, the sound output can also be connected, via the audio/video socket, to a hi-fi system for high quality playback or recording. In this instalment we look at the principles behind developing software that allows the Commodore 64 to be set up as a drum machine. The quality of the application is remarkably improved using playback through a hi-fi system.

In addition to the creation of complex sounds under software control, the SID chip can also accept input from an external audio signal. These signals may be generated by external electronic hardware (possibly containing other SID chips) or musical instruments such as electric guitars. This external signal can be mixed with the audio output of the SID chip and processed through its filters. However, we recommend caution when using I/O interfaces, since the incorrect connection of external lines can seriously damage the computer. We suggest that you consult the manufacturer's manual for the correct pinouts and levels before attempting any external connections.

To begin our discussion, we will consider how musical (periodic) sound can be made up from separate pure tones. We will then consider an alternative method of achieving the same result. By controlling the envelope of a given periodic note, we can introduce harmonics in different proportions. Effectively, this means we can create almost any periodic sound we like by altering a few simple 'hooks' on the envelope of an oscillator. These hooks — or controls — are also known as 'ADSR' (attack, decay, sustain and release).

Unfortunately, the BASIC resident in the Commodore 64 is not well-equipped to handle sound. Programming sound on the computer is achieved by extensive use of PEEK and POKE commands. If we set the variable SID equal to 54272 (the base address of the SID chip), then the addresses SID to SID+28 will control the sound chip — and hence all sound on the Commodore 64.

Finally, we give a machine code program to set up a three-voice, interrupt-driven drum machine. This wedge code can be controlled from BASIC, but will continue to play the drums independently of the BASIC program. This means that, with slight modifications, it can be used to provide background sound effects for any BASIC program.

THE SOUND OF MUSIC

Sound reaches our ears in the form of periodic vibrations of air pressure. The number of vibrations per second is called the 'pitch' — or 'frequency' — of the sound. The lower limit of human hearing is around 15 cycles per second (or 15 Hertz). A pure tone whose frequency is 100Hz sounds low to the ear. The A above middle C is standardised at 440Hz. If we double the frequency of a tone, then we raise its pitch by one octave. Normal human hearing spans a range of around 10.5 octaves. The three-tone oscillators of the SID chip, however, span a range of around eight octaves — approximately 0 to 4,000Hz.

Jean Fourier (1768-1830), a French physicist, first noticed that any periodic waveform can be considered to be composed of a fundamental pure tone amalgamated with tones of frequencies that are multiples of that fundamental tone. These tones are called 'harmonics'. What gives a note or sound its distinguishing timbre are the proportions in which the various harmonics appear.

A pure sine wave, which would correspond to a pure tone, is intrinsically an analogue signal, which means it is not an easy signal to produce on a digital device with voltage levels of either 0v or 5v. So, instead of generating pure frequencies and then mixing them to get the desired periodic sound, the approach adopted on home micros is different.

The SID chip is equipped with three voices, each of which can generate one of four different periodic waveforms. These are:

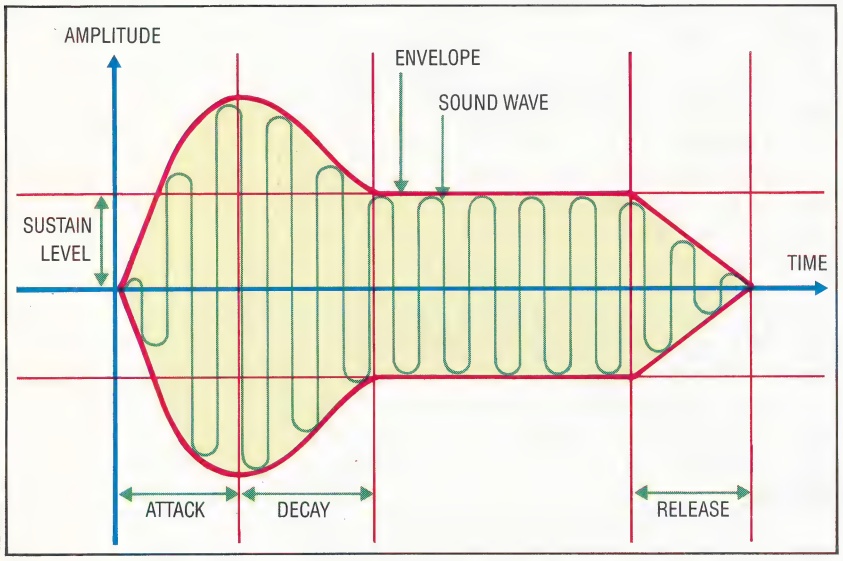
1) *Sawtooth Wave:* Here, all the harmonics are present. The Nth harmonic has an intensity proportional to 1/N.
2) *Triangular Wave:* This contains only odd harmonics. For an odd N, the Nth harmonic has an intensity proportional to $1/N^2$.
3) *Rectangular Wave:* A square wave has odd harmonics proportional to 1/N. By changing the 'pulse width', we can get a variety of rectangular waves, each having its own harmonic mix.
4) *White Noise:* A random mixture of frequencies, mostly used for special effects.

The harmonic content of a sound can also be altered by filtering. The SID chip offers three kinds of filter: low-pass, band-pass and high-pass. For example, a low-pass filter will pass all frequencies

below a preset value and attenuate all frequencies above the same value. By using these facilities and the ADSR envelope controls, almost any sound can be produced.

Speech-generating programs can be constructed in a variety of ways. One way, offering reasonable quality and unlimited vocabulary, is described here. In this method, the basic building blocks of speech — phonemes — are coded into an array of ADSR values. There are about 52 different phonemes in English, so encoding them is not a phenomenally difficult task. Then a machine code program is used to translate ASC (Commodore's ASCII alternative) text into a stream of phoneme codes, which can then be sent to the SID chip using the ADSR array. This is not quite so easy, primarily because the rules that are used to translate text to phonemes are quite complicated. The quality of the final speech will depend very heavily on how good this part of the program is. This is a perfectly feasible project for the Commodore 64 and a number of commercial products are available that use this technique.

ENVELOPE CONTROL



LIZ DIXON

Addressed Envelopes
The quality of a note — the group of characteristics that allow us to differentiate between, say, a piano and a violin — is dependent on the shape of the envelope. In the electronic synthesis of sounds the envelope is regarded as consisting of four distinct stages. These are known as attack, decay, sustain and release, or ADSR. The length of each stage of the ADSR envelope can be defined by POKEing values into the SID chip registers, allowing us to synthesise the sounds of different instruments on the Commodore 64

The ADSR diagram given here shows the general shape of a musical note with an emphasis on those features that the SID chip allows us to control. The four ADSR factors are:

1) *Attack:* The rise time of a note.
2) *Decay:* The time taken for a note to fall to a steady level.
3) *Sustain:* The volume of a steady level.
4) *Release:* The time taken for the volume of the note to fall from sustain level to zero.

The attack, decay and release sections of a note are each controlled by nybbles (four bits) within the SID chip registers. This means that each of these parameters takes a value between 0 and 15. The relationship between the values that must be POKEd into the SID registers and the real timings is given in the following table:

Value	Attack Rate (Time/Cycle)	Decay/Release Rate (Time/Cycle)
0	2 ms	6 ms
1	8 ms	24 ms
2	16 ms	48 ms
3	24 ms	72 ms
4	38 ms	114 ms
5	56 ms	168 ms
6	68 ms	204 ms
7	80 ms	240 ms
8	100 ms	300 ms
9	250 ms	750 ms
10	500 ms	1.5 secs
11	800 ms	2.4 secs
12	1 sec	3 secs
13	3 secs	9 secs
14	5 secs	15 secs
15	8 secs	24 secs

The length of the sustain section is determined by using a delay loop. Using this table, the ADSR values for a violin note might be:

ADSR	Time	POKE Value
A	500ms	10
D	300ms	8
S	—	—
R	750ms	9

To produce a sound on the Commodore 64 we have to take a minimum of six steps. For example:

- Step 1: Turn on the volume using:

```
POKE SID+24,15
```

- Step 2: Select ADSR. For example:

```
POKE SID+5,9 :REM ATTACK/DECAY VOICE #1
POKE SID+6,0 :REM SUSTAIN/RELEASE VOICE #1
```

- Step 3: Select the frequency for each oscillator. For example:

```
POKE SID+1,25:REM HI-BYTE FREQ VOICE#1
POKE SID,0    :REM LO-BYTE FREQ VOICE#1
```

- Step 4: Select the desired waveform type. For example:

```
POKE SID+4,33 :REM SELECT SAWTOOTH FOR
VOICE#1
```

At this point the sound begins to play — which is referred to as opening the 'gate'.

- Step 5: A delay loop while the note plays at the sustain level.

- Step 6: The waveform must be released. For example:

```
POKE SID+4,32 :REM RELEASE SAWTOOTH
```

The simplest way to program the Commodore 64 to play a tune is to set initial values of ADSR, and construct a FOR loop that READs frequency hi/lo bytes from DATA statements. By inserting zeros in the data statements, the tempo can be varied for different voices while keeping the delay constant.

Drum Machine

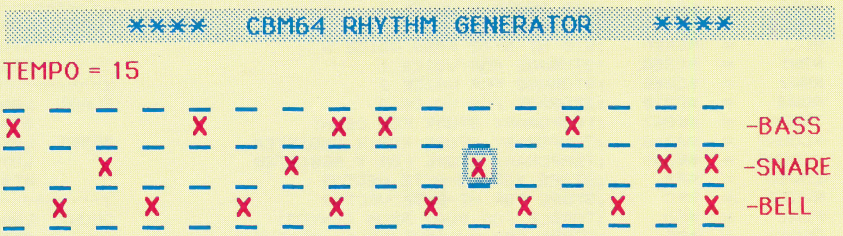
The Drum Machine program we give here uses an IRQ wedge, so that once the drums have been set up they will continue to play regardless of what we are doing in BASIC. The program also has an interesting way of checking from machine code to see what key has been pressed — it inspects the contents of location 197 ($00C5). This is a zero page location that contains a value describing the last key pressed. A machine code program can save you the trouble of a kernel call to get a keypress, provided $00C5 is sampled sufficiently often.

Note that because the BASIC driver does not actually empty the keyboard buffer using GET or INPUT, the program must set the pointer in location 198 ($00C6) to zero upon exit. This is the zero page location, which normally contains the number of keypresses stored in the keyboard buffer.

When using the program, each drum is allocated 16 time slots in which it can sound, and these are displayed in a grid on the screen. The choice of 16 makes it easy to set up a four-four rock rhythm. The sprite cursor is moved into one of the grid's cells, and when it is in the correct position a time slot can be activated/de-activated with the Return key, and the rhythm turned on using the f1 key. The on-screen menu displays the options available. Because the sounds are interrupt-driven, the rhythm pattern can be edited on screen while the rhythm is playing.

The BASIC program includes all the machine code necessary to drive the program as DATA statements and can be typed in and run as given. We also provide a source code listing for the machine code part of the program. If you wish to type in and assemble the source code, then lines 1520 to 1540 and the DATA statements between lines 1620 and 1940 can be omitted.

With the program running, try setting up the pattern below for a classic rhythm:



BASIC Rhythm Generator Program

```
1000 REM **RHYTHM GENERATOR**
1010 PRINTCHR$(147):REM CLEAR SCREEN
1020 GOSUB1510:REM SETUP M/C+SPRITE
1030 GOSUB1270:REM SETUP SCREEN
1040 SYS(49152):REM *INSERT WEDGE*
1050 REM SYS(49175) TO REMOVE WEDGE
1060 REM *MAIN LOOP*
1070 P=PEEK(197):REM CURRENT KEYPRESS
1080 IFP=37THENY=Y-16:SY=SY-1:IFY<95THENY=95:SY=0
1090 IFP=36THENY=Y+16:SY=SY+1:IFY>127THENY=127:SY=
2
1100 IFP=47THENX=X-16:SX=SX-1:IFX<28THENX=28:SX=0
1110 IFP=44THENX=X+16:SX=SX+1:IFX>268THENX=268:SX=
15
1120 Z=0:IFP=1THENZ=1
1130 PRINTCHR$(19);S$;"TEMPO =";PEEK(679);CHR$(157
);"   "
1140 POKEVIC+16,X/256:POKEVIC,XAND255
1150 POKEVIC+1,Y
1160 REM**COMPUTE SCREEN LOC FROM SPRITE X,Y POS
1170 SC=B+INT((Y-50)/8+1)*40+INT((X-18)/8)
1180 IFSY=0THENROW=50000:REM BASS DRUM
1190 IFSY=1THENROW=50016:REM SNARE DRUM
1200 IFSY=2THENROW=50032:REM COW BELL
1210 IFZ=1ANDPEEK(SC)=32THENPOKESC,24:POKEROW+SX,1
:GOTO1230
1220 IFZ=1ANDPEEK(SC)=24THENPOKESC,32:POKEROW+SX,0
1230 IFP=51THENGOSUB1270:FORI=0TO50:POKE50000+I,0:
NEXT
1240 IFP=57THENPOKE198,0:POKEVIC+21,0:PRINTCHR$(14
7):END
1250 GOTO1070
1260 REM *SET UP SCREEN*
1270 PRINTCHR$(19);
1280 S$=CHR$(17)+CHR$(17)+CHR$(17)
1290 PRINTTAB(3)"
"
1300 PRINTCHR$(18)TAB(3)" **** CBM64  RHYTHM GENER
ATOR **** "
1310 PRINT:PRINT:PRINT
1320 PRINT" - - - - - - - - - - - - - - - - "
1330 PRINT"                                  -BASS"

1340 PRINT" - - - - - - - - - - - - - - - - "
1350 PRINT"                                  -SNARE
"
1360 PRINT" - - - - - - - - - - - - - - - - "
1370 PRINT"                                  -BELL"

1380 PRINT" - - - - - - - - - - - - - - - - "
1390 PRINT:PRINTTAB(8)"OPTIONS...."
1400 PRINTTAB(8)"[F1]    PLAY RHYTHM"
1410 PRINTTAB(8)"[F3]    DECREASE TEMPO"
1420 PRINTTAB(8)"[F5]    INCREASE TEMPO"
1430 PRINTTAB(8)"[F7]    STOP RHYTHM"
1440 PRINTTAB(8)"[CLR]   CLEAR CHART"
1450 PRINTTAB(8)"[ST]     END PROGRAM"
1460 PRINTTAB(8)"[<]     MOVE CURSOR LEFT"
1470 PRINTTAB(8)"[>]     MOVE CURSOR RIGHT"
1480 PRINTTAB(8)"[K]     MOVE CURSOR UP"
1490 PRINTTAB(8)"[M]     MOVE CURSOR DOWN"
1500 RETURN
1510 REM *SET UP M/CODE & SPRITE*
1520 FORI=49152TO49413
1530 READJ:C=C+J:POKEI,J:NEXTI
1540 READJ:IFC<>JTHENPRINT"DATA ERROR":END
1550 FORI=0TO62:READJ:POKE832+I,J:NEXTI
1560 VIC=53248:X=28:Y=95:B=1024:SID=54272
1570 FORI=0TO24:POKESID+I,0:NEXTI
1580 POKESID+24,15:POKEVIC+21,1:POKE2040,13
1590 POKE254,15:POKE679,15:POKEVIC+39,1
1600 FORI=0TO50:POKE50000+I,0:NEXT
1610 RETURN
1620 REM***M/C DATA***
1630 DATA120,173,20,3,133,251,173,21,3
1640 DATA133,252,169,36,141,20,3,169
1650 DATA192,141,21,3,88,96,120,165,251
1660 DATA141,20,3,165,252,141,21,3,88
1670 DATA96,32,177,192,165,197,201,3
1680 DATA208,3,32,135,192,165,197,201,4
1690 DATA208,3,32,140,192,165,197,201,5
1700 DATA208,3,32,155,192,165,197,201,6
1710 DATA208,3,32,166,192,32,87,192,142
1720 DATA168,2,140,169,2,76,49,234,165
1730 DATA253,208,1,96,198,254,240,1,96
1740 DATA32,149,192,224,16,208,3,32,144
1750 DATA192,185,80,195,240,3,32,184
1760 DATA192,185,96,195,240,3,32,210
1770 DATA192,185,112,195,240,3,32,236
1780 DATA192,200,234,232,96,169,0,133
1790 DATA253,96,169,1,133,253,162,0,160
1800 DATA0,96,173,167,2,133,254,96,173
1810 DATA167,2,201,255,240,3,238,167,2
1820 DATA96,173,167,2,201,1,240,3,206
1830 DATA167,2,96,174,168,2,172,169,2
1840 DATA96,169,14,141,6,212,169,32,141
1850 DATA2,212,169,66,141,4,212,169,3
1860 DATA141,1,212,169,65,141,4,212,96
1870 DATA169,7,141,12,212,169,12,141,13
1880 DATA212,169,128,141,11,212,169,65
1890 DATA141,8,212,169,129,141,11,212
1900 DATA96,169,2,141,19,212,169,13,141
1910 DATA20,212,169,18,141,18,212,169
1920 DATA100,141,15,212,169,17,141,18
1930 DATA212,96
1940 DATA32038:REM*CHECKSUM*
1950 REM**SPRITE CURSOR DATA**
1960 DATA127,254,0,127,254,0,127,254,0
1970 DATA112,14,0,112,14,0,112,14,0,112
1980 DATA14,0,112,14,0,112,14,0,112,14
1990 DATA0,112,14,0,127,254,0,127,254,0
2000 DATA127,254,0,0,0,0,0,0,0,0,0,0,0
2010 DATA0,0,0,0,0,0,0,0,0,0,0
```

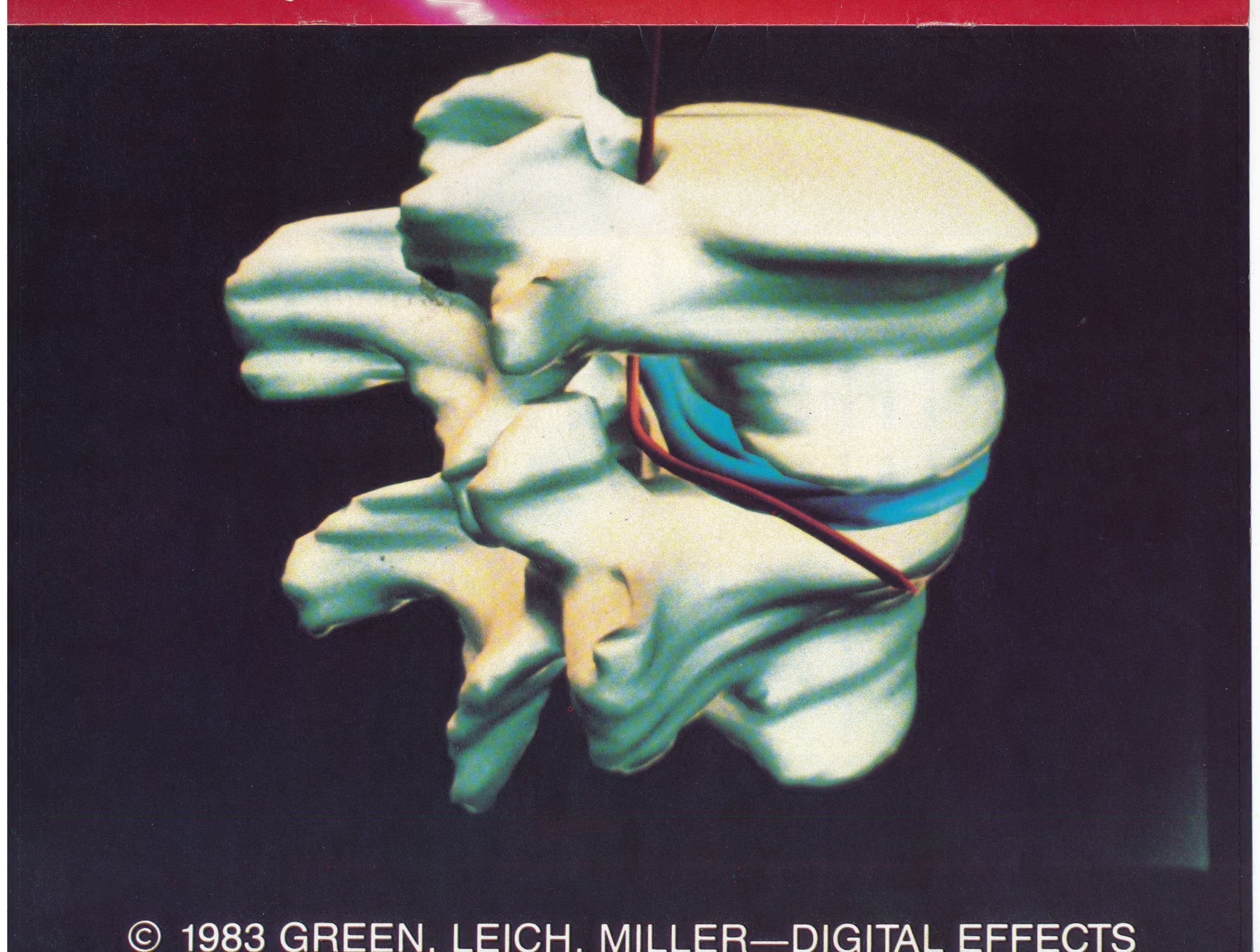
Assembly Listing

```
;++++++++++++++++++++++++
;++++++++++++++++++++++++
;++                    ++
;++  CBM DRUM MACHINE ++
;++    SOURCE CODE     ++
;++                    ++
;++++++++++++++++++++++++
;++++++++++++++++++++++++
;
VOL     =  $D418    ;SID VOLUME
ATT1    =  $D405    ;SID ATTACK VOICE 1
SUS1    =  $D406    ;SID SUSTAIN VOICE 1
PULSE   =  $D402    ;SID PULSE RATE VOICE 1
WAVE1   =  $D404    ;SID WAVEFORM VOICE 1
BASS    =  $D401    ;SID HI-BYTE FREQ VOICE 1
ATT2    =  $D40C    ;SID ATTACK VOICE 1
SUS2    =  $D40D    ;SID SUSTAIN VOICE 2
WAVE2   =  $D40B    ;SID WAVEFORM VOICE 2
SNARE   =  $D408    ;SID HI-BYTE FREQ VOICE 2
ATT3    =  $D413    ;SID ATTACK VOICE 3
SUS3    =  $D414    ;SID SUSTAIN VOICE 3
WAVE3   =  $D412    ;SID WAVEFORM VOICE 3
BELL    =  $D40F    ;SID HI-BYTE FREQ VOICE 3
ROW1    =  $C350    ;STORAGE FOR VOICE 1
ROW2    =  $C360    ;STORAGE FOR VOICE 2
ROW3    =  $C370    ;STORAGE FOR VOICE 3
TEMPO   =  $02A7    ;TEMPORARY STORE FOR DELAY
XCOUNT  =  $02A8    ;TEMPORARY STORE FOR X REGISTER
YCOUNT  =  $02A9    ;TEMPORARY STORE FOR Y REGISTER
LOVEC   =  $FB      ;STORE FOR LO-BYTE VECTOR
HIVEC   =  $FC      ;STORE FOR HI-BYTE VECTOR
PLAY    =  $FD      ;TOGGLE FOR PLAY DRUM (1=YES)
DELAY   =  $FE      ;STORE FOR CURRENT DELAY STATUS
KEY     =  $C5      ;CURRENT KEY PRESSED
;-----------------------------------------------------
*       =  $C000    ;ASSEMBLE FROM 49152 (DECIMAL)
;-----------------------------------------------------
;SET UP WEDGE
;-----------------------------------------------------
        SEI         ;DISABLE INTERRUPT REQUEST
        LDA $0314   ;GET CONTENTS OF VECTOR LO-BYTE
        STA LOVEC   ;STORE IN LOVEC
        LDA $0315   ;GET CONTENTS OF VECTOR HI-BYTE
        STA HIVEC   ;STORE IN HIVEC
        LDA #<WEDGE ;GET LO-BYTE WEDGE START ADDR.
        STA $0314   ;STORE IN IRQ VECTOR LO-BYTE
        LDA #>WEDGE ;GET HI-BYTE OF WEDGE START
        STA $0315   ;STORE IN IRQ VECTOR HI-BYTE
        CLI         ;RESUME INTERRUPT REQUEST
        RTS         ;RETURN
;-----------------------------------------------------
;REMOVE WEDGE
;-----------------------------------------------------
        SEI         ;DISABLE INTERRUPT REQUEST
        LDA LOVEC   ;GET ORIGINAL VALUE FROM LOVEC
        STA $0314   ;STORE IN IRQ VECTOR LO-BYTE
        LDA HIVEC   ;GET ORIGINAL VALUE FROM HIVEC
        STA $0315   ;STORE IN IRQ VECTOR HI-BYTE
        CLI         ;RESUME INTERRUPT REQUEST
        RTS         ;RETURN
;-----------------------------------------------------
;MAIN LOOP
;-----------------------------------------------------
WEDGE   JSR REG     ;GOSUB REG
        LDA KEY     ;WHICH KEY PRESSED ?
        CMP #$03    ;IS IT FUNCTION KEY #1 ?
        BNE CONT1   ;BRANCH IF NOT TRUE
        JSR FLAG0   ;GOSUB FLAG0
CONT1   LDA KEY     ;WHICH KEY PRESSED ?
        CMP #$04    ;IS IT FUNCTION KEY #7 ?
        BNE CONT2   ;BRANCH IF NOT TRUE
        JSR FLAG1   ;GOSUB FLAG1
CONT2   LDA KEY     ;WHICH KEY PRESSED ?
        CMP #$05    ;IS IT FUNCTION KEY #3 ?
        BNE CONT3   ;BRANCH IF NOT TRUE
        JSR ADD     ;GOSUB ADD
CONT3   LDA KEY     ;WHICH KEY PRESSED ?
        CMP #$06    ;IS IT FUNCTION KEY #5 ?
        BNE CONT4   ;BRANCH IF NOT TRUE
        JSR MINUS   ;GOSUB MINUS
CONT4   JSR REST    ;GOSUB REST
        STX XCOUNT  ;STORE VALUE OF X REGISTER
        STY YCOUNT  ;STORE VALUE OF Y REGISTER
        JMP $EA31   ;JUMP BACK TO INTERRUPT
;-----------------------------------------------------
;CHECK ROUTINE
;-----------------------------------------------------
REST    LDA PLAY    ;GET TOGGLE VALUE
        BNE BEGIN   ;IF EQUAL TO 1 THEN BRANCH
        RTS         ;RETURN
;-----------------------------------------------------
BEGIN   DEC DELAY   ;DECREMENT DELAY
        BEQ START   ;BRANCH IF ZERO
        RTS         ;RETURN
;-----------------------------------------------------
START   JSR COUNT   ;GOSUB COUNT
        CPX #$10    ;END OF LOOP ?
        BNE CHECK   ;BRANCH IF NOT 0
        JSR RESET   ;GOSUB RESET
;-----------------------------------------------------
CHECK   LDA ROW1,Y  ;GET VALUE OF ROW1 OFFSET BY Y
        BEQ NEXT1   ;BRANCH IF ZERO
        JSR DRUM1   ;GOSUB DRUM1
NEXT1   LDA ROW2,Y  ;GET VALUE OF ROW2 OFFSET BY Y
        BEQ NEXT2   ;BRANCH IF ZERO
        JSR DRUM2   ;GOSUB DRUM2
NEXT2   LDA ROW3,Y  ;GET VALUE OF ROW3 OFFSET BY Y
        BEQ NEXT3   ;BRANCH IF ZERO
        JSR DRUM3   ;GOSUB DRUM3
NEXT3   INY         ;INCREMENT OFFSET
        INX         ;INCREMENT LOOP COUNTER
        RTS         ;RETURN
;-----------------------------------------------------
;SUBROUTINES
;-----------------------------------------------------
FLAG0   LDA #$00    ;STORE ZERO -
        STA PLAY    ;IN PLAY
        RTS         ;RETURN.
;-----------------------------------------------------
FLAG1   LDA #$01    ;STORE 1 -
        STA PLAY    ;IN PLAY
RESET   LDX #$00    ;RESET X REGISTER
        LDY #$00    ;RESET Y REGISTER
        RTS         ;RETURN
;-----------------------------------------------------
COUNT   LDA TEMPO   ;GET VALUE OF TEMPO
        STA DELAY   ;STORE IN DELAY
        RTS         ;RETURN
;-----------------------------------------------------
ADD     LDA TEMPO   ;GET VALUE OF TEMPO
        CMP #$FF    ;COMPARE RESULT WITH 255
        BEQ CONT5   ;BRANCH IF TRUE
        INC TEMPO   ;INCREMENT TEMPO
CONT5   RTS         ;RETURN
;-----------------------------------------------------
MINUS   LDA TEMPO   ;GET VALUE OF TEMPO
        CMP #$01    ;COMPARE RESULT WITH 1
        BEQ CONT6   ;BRANCH IF TRUE
        DEC TEMPO   ;DECREMENT TEMPO
CONT6   RTS         ;RETURN
;-----------------------------------------------------
REG     LDX XCOUNT  ;STORE VALUE OF XCOUNT IN X REG
        LDY YCOUNT  ;STORE VALUE OF YCOUNT IN Y REG
        RTS         ;RETURN
;-----------------------------------------------------
;DRUM SOUND ROUTINES
;-----------------------------------------------------
DRUM1   LDA #$0E    ;SET
        STA SUS1    ;
        LDA #$20    ;UP
        STA PULSE   ;
        LDA #$42    ;BASS
        STA WAVE1   ;
        LDA #$03    ;DRUM
        STA BASS    ;
        LDA #$41    ;AND PLAY
        STA WAVE1   ;
        RTS         ;RETURN
;-----------------------------------------------------
DRUM2   LDA #$07    ;SET
        STA ATT2    ;
        LDA #$0C    ;UP
        STA SUS2    ;
        LDA #$80    ;SNARE
        STA WAVE2   ;
        LDA #$41    ;DRUM
        STA SNARE   ;
        LDA #$81    ;AND PLAY
        STA WAVE2   ;
        RTS         ;RETURN
;-----------------------------------------------------
DRUM3   LDA #$02    ;SET
        STA ATT3    ;
        LDA #$0D    ;UP
        STA SUS3    ;
        LDA #$12    ;COW
        STA WAVE3   ;
        LDA #$64    ;BELL
        STA BELL    ;
        LDA #$11    ;AND PLAY
        STA WAVE3   ;
        RTS         ;RETURN
        .END
```

DATABASE

Here, courtesy of Commodore Business Machines, we publish another instalment of a detailed analysis of the Commodore 64's memory map.

LABEL	HEX ADDRESS	DECIMAL LOCATION	DESCRIPTION
LASTPT	0017–0018	23–24	Last Temp String Address
TEMPST	0019–0021	25–33	Stack for Temporary Strings
INDEX	0022–0025	34–37	Utility Pointer Area
RESHO	0026–002A	38–42	Floating-Point Product of Multiply
TXTTAB	002B–002C	43–44	Pointer: Start of BASIC Text
VARTAB	002D–002E	45–46	Pointer: Start of BASIC Variables
ARYTAB	002F–0030	47–48	Pointer: Start of BASIC Arrays
STREND	0031–0032	49–50	Pointer: End of BASIC Arrays (+1)
FRETOP	0033–0034	51–52	Pointer: Bottom of String Storage
FRESPC	0035–0036	53–54	Utility String Pointer
MEMSIZ	0037–0038	55–56	Pointer: Highest Address Used by BASIC
CURLIN	0039–003A	57–58	Current BASIC Line Number
OLDLIN	003B–003C	59–60	Previous BASIC Line Number
OLDTXT	003D–003E	61–62	Pointer: BASIC Statement for CONT
DATLIN	003F–0040	63–64	Current DATA Line Number
DATPTR	0041–0042	65–66	Pointer: Current DATA Item Address
INPPTR	0043–0044	67–68	Vector: INPUT Routine
VARNAM	0045–0046	69–70	Current BASIC Variable Name
VARPNT	0047–0048	71–72	Pointer: Current BASIC Variable Data
FORPNT	0049–004A	73–74	Pointer: Index Variable for FOR/NEXT
	004B–0060	75–96	Temp Pointer / Data Area
FACEXP	0061	97	Floating-Point Accumulator #1: Exponent



© 1983 GREEN, LEICH, MILLER—DIGITAL EFFECTS